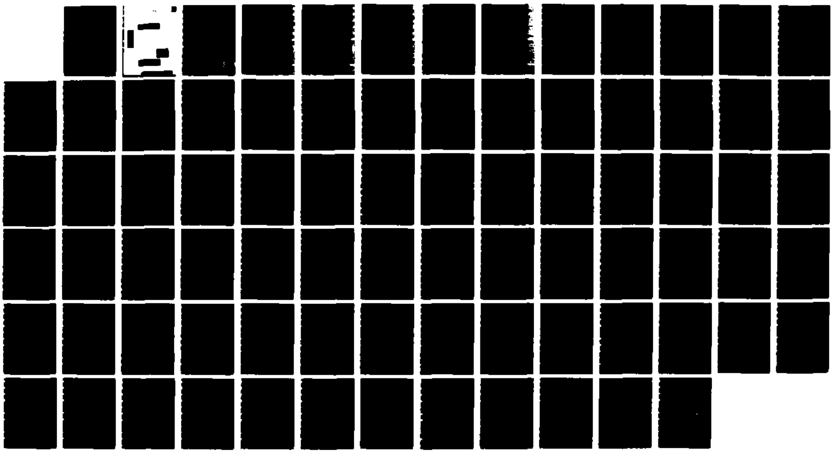
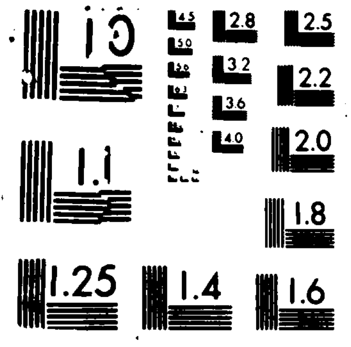
AD-A182 178 DATA REPLICATION IN NESTED TRANSACTION SYSTEMS(U) 1/1
MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER
SCIENCE K J GOLDMAN MAY 87 MIT/LCS/TR-390
UNCLASSIFIED N00014-85-K-0168 F/G 12/7 NL




1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

DATA REPLICATION IN NESTED TRANSACTION SYSTEMS

Kenneth J. Goldman

May 1987

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE A182 172

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR-390	5 MONITORING ORGANIZATION REPORT NUMBER(S) DOD/DARPA # N00014-85-K-0168, N00014-83-K-0125
6a NAME OF PERFORMING ORGANIZATION Laboratory for Computer Science 6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION Office of Naval Research/Department of Navy
6c. ADDRESS (City, State, and ZIP Code) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (City, State, and ZIP Code) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22217	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
Data Replication in Nested Transaction Systems

12 PERSONAL AUTHOR(S)
Goldman, Kenneth J.

13a. TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Technical	FROM ______ TO ______	1987 May	79

16. SUPPLEMENTARY NOTATION

17 COSATI CODES		
FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
data replication, nested transaction systems, input-output automata, models of distributed computation

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

Giffords's basic Quorum Consensus algorithm for data replication is generalized to accomodate nested transactions and transaction failures (aborts). A formal description of the generalized algorithm is presented using the new Lynch-Merritt input-ouput automaton model for nested transaction systems. This formal description is used to construct a complete (yet simple) proof of correctness that uses standard assertional techniques and is based on a natural correctness condition. Nondeterminism is used in the algorithm description to yield a correctness proof that is independent of any particular programming language or implementation. The presentation and proof treat issues of data replication entirely separately from issues of concurrency control and recovery.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Judy Little, Publications Coordinator	22b TELEPHONE (Include Area Code) (617) 253-5894 22c OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete
SECURITY CLASSIFICATION OF THIS PAGE
*U.S. Government Printing Office: 1985-507-047
Unclassified

Data Replication in Nested Transaction Systems

by

Kenneth J. Goldman
Sc.B., Brown University
(December, 1984)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE
IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1987

Signature of Author ______________________
Department of Electrical Engineering and Computer Science
May 6, 1987

Certified by ______________________
Nancy A. Lynch
Thesis Supervisor

Accepted by ______________________
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Data Replication in Nested Transaction Systems

by

Kenneth J. Goldman

Submitted to the
Department of Electrical Engineering and Computer Science
on May 6, 1987, in partial fulfillment of the requirements
for the Degree of
Master of Science in Computer Science

Abstract

Gifford's basic Quorum Consensus algorithm for data replication is generalized to accommodate nested transactions and transaction failures (aborts). A formal description of the generalized algorithm is presented using the new Lynch-Merritt input-output automaton model for nested transaction systems. This formal description is used to construct a complete (yet simple) proof of correctness that uses standard assertional techniques and is based on a natural correctness condition. Nondeterminism is used in the algorithm description to yield a correctness proof that is independent of any particular programming language or implementation. The presentation and proof treat issues of data replication entirely separately from issues of concurrency control and recovery.

Keywords: data replication, nested transaction systems, input-output automata, models of distributed computation.

Thesis supervisor: Nancy A. Lynch
Title: Ellen Swallow Richards Professor of Computer Science and Engineering

Acknowledgments

This thesis would not have been possible without all of the people who gave me help and encouragement.

I am grateful to my advisor, Nancy Lynch, for suggesting this topic and for providing many useful insights. I am also thankful for her patience in teaching me the importance of rigor and detail, and for her careful reading of many drafts of this thesis.

The members of the Theory of Distributed Systems group, Brian Coan, Alan Fekete, Mark Tuttle, and Jennifer Welch, were generous with their time in giving advice and technical help. In particular, I thank Alan Fekete and Jennifer Welch for their comments on earlier drafts. Also, I would like to thank Bill Weihl and Sharon Perl for helpful discussions during the early stages of this work.

My wife, Sally, has not only provided useful technical comments on this thesis, but also has been an endless source of emotional support. Her understanding and encouragement have contributed both to the quality of this project and to my pleasure in working on it.

This work was supported in part by the Office of Naval Research under Contract N00014-85-K-0168, by the Office of Army Research under Contract DAAG29-84-K-0058, by the National Science Foundation under Grants DCR-83-02391 and CCR-8611442, and by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125.

Kenneth Goldman was supported by an Office of Naval Research graduate fellowship.

Contents

1 Introduction 9

2 The Model 13
- 2.1 I/O Automata and Systems 13
- 2.2 Nested Transaction Systems 16
- 2.3 Model Extensions for Replicated Data Systems 22

3 Fixed Quorum Consensus 26
- 3.1 Replicated Serial System 26
- 3.2 Non-replicated Serial System 38
- 3.3 Correctness 40
- 3.4 Concurrent Replicated Systems 43

4 Reconfigurable Quorum Consensus 44
- 4.1 Reconfigurable Replicated Serial System 46
  - 4.1.1 Data Managers 47
  - 4.1.2 Coordinators 49
  - 4.1.3 Transaction Managers 53
  - 4.1.4 Spy Automata 58
  - 4.1.5 Properties 60
- 4.2 Non-replicated Serial System 69
- 4.3 Correctness 71
- 4.4 Concurrent Replicated Systems 74

5 Conclusion 76

Chapter 1

Introduction

In distributed database systems, logical data items are often *replicated* in order to improve availability, reliability and performance. Whenever replication is used, a *replication algorithm* is required in order to ensure that the replication is transparent to the user programs. In understanding replication algorithms, it is convenient to think of each logical data item as being implemented by a collection of data managers (DMs) and transaction managers (TMs). The DMs retain state information, and the collective state of the DMs defines the current state of the logical data item. The user programs invoke TMs in order to read or write the logical data item; the TMs accomplish this by physically accessing some subset of the DMs.

One of the most well-known replication algorithms is Gifford's algorithm [Gi], which we call *Quorum Consensus*. Based on Thomas [T], the ideas of this method underlie many of the more recent and sophisticated replication techniques (e.g., [ASC,AT,ES,He]). In Gifford's algorithm, each DM is assigned a certain number of *votes* and keeps as part of its state a data *value* with an associated *version number*. Each logical data item x has an associated *configuration* that consists of a pair of integers called *read-quorum* and *write-quorum*. If v is the total number of votes assigned to DMs for x, then the configuration is constrained so that read-quorum + write-quorum $> v$. To read x, a TM collects the version-numbers and values from enough DMs so that it has a read-quorum of votes; then it returns the value associated with the highest version-number. To write x, a TM first collects the version-numbers from enough DMs so that it has a read-quorum of votes; then, it writes its value with a higher version number to a collection of DMs with a write-quorum of votes. This method generalizes both the read-one/write-all and the read-majority/write-majority algorithms.

Here, we adopt a slightly more general configuration strategy, which is justified by Barbara and Garcia-Molina in [BaGa]: A configuration consists of a set of read-quorums and a set of write-quorums. Each quorum is a set of DM names, and every read-quorum must have a non-empty intersection with every write-quorum. To read a data item, a TM accesses all the DMs in some read-quorum and chooses the value with the highest version number. To write a data item, a TM first discovers the highest version number written so far by accessing all the DMs in some read-quorum; then the TM increments that version number by one and writes the new value and version number to all the DMs in some write-quoı um.

In this thesis, we generalize Gifford's algorithm in three fundamental ways. First, we incorporate the concept of *transaction nesting* into the algorithm. Transaction nesting is useful in its own right (for instance, as the basis of the distributed programming language ARGUS [LHJLSW,LiS,Mo,We]). In addition, it turns out that nested transactions provide a useful way of understanding replication algorithms even if user transactions are not nested (as in Gifford [Gi]). This is because the TM's themselves can be regarded as *subtransactions* of the user transactions. Once one sees how to understand the algorithm in this way, it is very natural to generalize the algorithm to allow nesting of user transactions as well. Second, we extend the algorithm to accommodate transaction failures (aborts). Thus, for example, an operation to access a logical data item can complete even if some of its associated DM accesses abort. Finally, we provide a fully-developed version of the mechanism outlined by Gifford for changing the read- and write-quorums dynamically. This capability, known as *reconfiguration*, generally is used for coping with site or link failures. We obtain a single algorithm that integrates all three generalizations.

We present our algorithm using the new framework of Lynch and Merritt [LM] for modeling nested transaction concurrency control and recovery. For clarity, we present two versions of the algorithm. In the first version, we assume that the configuration for each data item is fixed and is known in advance to all the TMs that access that data item. The second version includes the reconfiguration mechanism. The descriptions are clear, simple, and unambiguous. A complete correctness proof is also included; it is short, natural, and intuitive, yet completely rigorous.

An important reason for the simplicity of the proof is the fact that we are able to separate the treatment of replication entirely from the treatment of concurrency control and recovery. That is, we are able to consider the replication issues solely in the context of serial systems. We prove that a system which includes the new replication algorithm and which is serial at the level of the individual data copies "simulates" (in a strong sense) a system which is serial at the level of the logical data items. In particular, it "looks the same" to the user transactions. Since both systems involved in this simulation are serial systems, the simulation proof is very simple, and is based on standard assertional techniques.

Of course, systems which are truly serial at the level of the data copies are of little practical interest. However, previous work on nested transaction concurrency control and recovery algorithms [Mo,R,LM,FLMW] has produced several interesting algorithms which guarantee that a system *appears to be serial*, as far as the transactions can tell. Combining any of these algorithms (at the copy level) with the new replication algorithm yields a combined algorithm which appears to be non-replicated and serial (at the logical data item level), as far as the user transactions can tell.

In fact, our results show that the replication algorithm can be combined with *any* algorithm which guarantees "serializability" at the copy level, to yield a system which is serializable at the logical item level. Thus, our work formalizes a frequently stated informal claim that "quorum consensus works with any correct concurrency control algorithm. As long as the algorithm produces serializable executions, quorum consensus will ensure that the effect is just like an execution on a single copy database" [BHG].

The presentation and proof techniques presented here work so well that we expect they will be of general use in simplifying the treatment of many other data replication algorithms as well.

Related work, in addition to the papers already mentioned, includes some previous attempts at rigorous presentation and proof of replicated data algorithms. Most notable among these is the presentation and proof given by Bernstein, Hadzilacos, and Goodman [BHG] of Gifford's basic algorithm. This work is based on ***serializability theory***, a theory which has made a significant contribution to the understanding of concurrency control. This approach, however, does not appear to generalize easily to the case where nesting and failures are allowed. Also, Herlihy [He] extends Gifford's algorithm to accommodate abstract data types and offers a correctness proof. Again, nesting is not considered. This thesis is part of a larger effort to unify the work in concurrency control and recovery, as well as extend it to permit nesting [LM,FLMW,HLMW].

The remainder of the thesis is organized as follows. In Chapter 2, we introduce the computation model. Then, in Chapter 3, we describe the generalized version of Gifford's algorithm without reconfiguration and prove its correctness. In Chapter 4, we expand on these results to give the description and correctness proof of the complete algorithm (i.e., including reconfiguration). For both versions of the algorithm, we show that the correctness of interesting non-serial replicated systems follows directly from these results. Chapter 5 contains a summary of our results and a brief discussion of possible further research.

Chapter 2

The Model

We use the I/O automaton model, due to Lynch-Merritt [LM] and Lynch-Tuttle [LT], as the formal foundation for our work. We model components of a system by (possibly infinite-state) nondeterministic automata that have operation names associated with their state transitions. Communication among automata is described by identifying their operations. We only prove properties of finite behavior, so a simple special case of the general model is sufficient. Sections 2.1 and 2.2 provide a brief introduction to I/O automata and systems that includes the definitions from [LM] and [LT] that are relevant to this work. Then, in Section 2.3, we extend the model with some new definitions that are particularly useful for modeling replicated data management algorithms.

2.1 I/O Automata and Systems

The basic components of the model are *I/O automata*. An I/O automaton $\mathcal{A}$ has components *states*($\mathcal{A}$), *start*($\mathcal{A}$), *out*($\mathcal{A}$), *in*($\mathcal{A}$), and *steps*($\mathcal{A}$). Here, *states*($\mathcal{A}$) is a set of states, of which a subset *start*($\mathcal{A}$) is designated as the set of start states. The next two components are disjoint sets: *out*($\mathcal{A}$) is the set of *output operations*, and *in*($\mathcal{A}$) is the set of *input operations*. The union of these two sets is the set of *operations* of the automaton. Finally, *steps*($\mathcal{A}$) is the transition relation of $\mathcal{A}$, which is a set of triples of the form (s', π, s), where s' and s are states, and π is an operation. This triple means that in state s', the automaton can atomically perform operation π and change to state s. An element of the transition relation is called a ***step*** of $\mathcal{A}$. If (s', π, s) is a step of $\mathcal{A}$, we say that π is ***enabled*** in s'.

The output operations are intended to model the actions that are triggered by the automaton itself, while the input operations model the actions that are triggered by the environment of the automaton. We require the following condition, which says that an I/O automaton must be prepared to receive any input operation at any time.

Input Condition: For each input operation π and each state s', there exist a state s and a step (s', π, s).

An *execution* of $\mathcal{A}$ is a finite alternating sequence $s_0, \pi_1, s_1, \pi_2, ..., s_n$ of states and operations of $\mathcal{A}$, where s_0 is in $start(\mathcal{A})$ and each subsequence $(s_i, \pi_{i+1}, s_{i+1})$ is in $steps(\mathcal{A})$. From any execution, we can extract the *schedule*, which is the subsequence of the execution that contains only the operations (e.g., $\pi_1, \pi_2, ..., \pi_n$). Because transitions to different states may have the same operation, different executions may have the same schedule.

If S is any set of schedules (or *property* of schedules), then automaton $\mathcal{A}$ is said to *preserve* S provided that the following holds. If $\alpha = \alpha'\pi$ is any schedule of $\mathcal{A}$, where π is an output operation and α' is in S, then α is in S. That is, $\mathcal{A}$ is not the first to violate the property described by S.

We model a *system* as a set of interacting components, each of which is an I/O automaton. It is convenient and natural to view systems as I/O automata as well. Thus, we define an operation that composes a set of I/O automata to yield a new I/O automaton.

A set of I/O automata may be composed to create a *system* $\mathcal{S}$, provided that the sets of output operations for the automata are disjoint. Thus, every output operation in $\mathcal{S}$ will be triggered by exactly one component. The system $\mathcal{S}$ is itself an I/O automaton. A state of the composed automaton is a tuple of states, one for each component, and the *start states* are tuples consisting of start states of the components. The set of *operations* of $\mathcal{S}$, $ops(\mathcal{S})$, is the union of the sets of operations of the component automata. The set of *output operations* of $\mathcal{S}$, $out(\mathcal{S})$, is likewise the union of the sets of output operations of the component automata. Finally, the set of *input operations* of $\mathcal{S}$, $in(\mathcal{S})$, is $ops(\mathcal{S}) - out(\mathcal{S})$, the set of operations of $\mathcal{S}$

that are not output operations of S. The output operations of a system are intended to be exactly those that are triggered by components of the system, while the input operations of a system are those that are triggered by the system's environment.

The triple (s', π, s) is in the transition relation of S if and only if for each component automaton A, one of the following two conditions holds. Either π is an operation of A, and the projection of the step onto A is a step of A, or else π is not an operation of A, and the state corresponding to A in tuple s' is identical to the state corresponding to A in tuple s. Thus, each operation of the composed automaton is an operation of a subset of the component automata. During the performance of an operation π of S, each of the components which has operation π carries out the operation, while the remainder stay in the same state. Again, the operation π is an output operation of the composition if it is the output operation of a component — otherwise, π is an input operation of the composition.

An *execution* of a system is defined to be an execution of the composition of the automata modeling the individual system components. If σ is a sequence of operations of a system S with component A, then $\sigma|A$ (read "σ restricted to A") is the subsequence of σ containing exactly the operations of A. Clearly, if σ is a schedule of S, then $\sigma|A$ is a schedule of A.

The following lemma, known as the Composition Lemma, expresses formally the notion that an operation is under the control of the component of which it is an output.

Lemma 1 Let σ' be a schedule of a system S, and let $\sigma = \sigma'\pi$, where π is an output operation of component A. If $\sigma|A$ is a schedule of A, then σ is a schedule of S.

Proof: In [LM]. ∎

Let σ be a schedule of system S. We say that property P *holds after* σ iff property P holds for the final state of every execution of S whose schedule is σ. We say that property P *holds forever after* σ iff property P holds for the final state of every execution of S whose schedule has σ as a prefix.

Let A be an automaton whose transition relation is restricted so that if (s', π, s_1) and (s', π, s_2) are both in steps(A), then $s_1 = s_2$. If A has a unique initial state, then we say that A is a *state-deterministic* automaton. That is, A is deterministic in the sense that its state is a function of its schedule.

All of the automata that we define explicitly are state-deterministic. For such automata, we will freely use the words "***state*** s ***of*** $\mathcal{A}$ ***after schedule*** σ" to denote the unique state of $\mathcal{A}$ resulting from the execution of $\mathcal{A}$ whose schedule is σ.

2.2 Nested Transaction Systems

To model nested transaction systems we use a ***system type***, which is a tuple ($\mathcal{T}$,parent,$\mathcal{O}$,V). $\mathcal{T}$ is the set of transaction names organized into a tree by the mapping parent:$\mathcal{T} \rightarrow \mathcal{T}$, with T_0 as the root. In referring to this tree, we use traditional terminology, such as child, leaf, least common ancestor (lca), ancestor and descendant. (A transaction is its own ancestor and descendant.) The leaves of $\mathcal{T}$ are called ***accesses***. The set $\mathcal{O}$ is a partition of the set of accesses, where each element (class) of the partition contains the accesses to a particular object; each element of $\mathcal{O}$ denotes its corresponding object. Finally, V is the set of ***values*** that may be returned by transactions. The tree structure is known in advance by all the components of the system and can be thought of as a predefined naming scheme for all possible transactions that might ever be invoked. In general, the tree is an infinite structure, and only some of the transactions will take steps in any given execution.

The root transaction T_0 plays a special role in this theory. The root models the environment of the nested transaction system (the "external world") from which requests for transactions originate and to which the results of these transactions are reported. Since it has no parent, T_0 may neither commit nor abort. The classical transactions of concurrency control theory (without nesting) appear in our model as the children of T_0. (In other work on nested transactions, such as Argus, the children of T_0 are often called "top-level" transactions.) Even in the context of classical theory (with no additional nesting) it is convenient to introduce the root transaction to model the environment in which the rest of the transaction system runs, with operations that describe the invocation and return of the classical transactions. It is natural to reason about T_0 in the same way as about all of the other transactions.

The internal nodes of the tree model transactions whose function is to create and manage subtransactions, but not to access data directly. The only transactions which actually access data are the leaves of the transaction tree, and thus they are called "accesses". The partition $\mathcal{O}$ simply identifies those transactions which access the same object.

The systems we describe are *serial systems*. A serial system is the composition of a set of I/O automata. This set contains a *transaction* for each internal node of the transaction tree, a *basic object* for each element of $\mathcal{O}$, and a *serial scheduler* for the given system type. The system *primitives* are the transaction automata and the basic objects; these describe user programs and data, respectively. The serial scheduler controls communication between the primitives, and thereby defines the allowable orders in which the primitives may take steps. All three types of system components are modelled as I/O automata. These automata are described below. (If X is a basic object associated with an element $\mathcal{X}$ of the partition $\mathcal{O}$, and T is an access in $\mathcal{X}$, we write T∈*accesses*(X) and say that "T is an access to X".)

Non-access Transactions: Transactions are modelled as I/O automata. In modeling transactions, we consider it very important not to constrain them unnecessarily; thus, we do not want to require that they be expressible as programs in any particular high-level programming language. Modeling the transactions as I/O automata allows us to state exactly the properties that are needed, without introducing unnecessary restrictions or complicated semantics.

A non-access transaction T is modelled as an I/O automaton, with the following operations:

Input operations:	CREATE(T)
	COMMIT(T',v), where T'∈children(T) and v∈ V
	ABORT(T'), where T'∈children(T)
Output operations:	REQUEST-CREATE(T'), where T'∈children(T)
	REQUEST-COMMIT(T,v), where v∈ V

The CREATE input operation "wakes up" the transaction. The REQUEST-CREATE output operation is a request by T to create a particular child transaction [1]. The COMMIT

[1]Note that there is no provision for T to pass information to its child in this request. In a programming language, T might be permitted to pass parameter values to a subtransaction. Although this may be a convenient descriptive aid, it is not necessary to include in it the underlying formal model. Instead, we consider transactions that have different input parameters to be different transactions.

input operation reports to T the successful completion of one of its children, and returns a value recording the results of that child's execution. The ABORT input operation reports to T the unsuccessful completion of one of its children, without returning any other information. We call COMMIT(T',v), for any v, and ABORT(T') *return* operations for transaction T'. The REQUEST-COMMIT operation is an announcement by T that it has finished its work, and includes a value for reporting the results of that work to its parent.

It is convenient to use two separate operations, REQUEST-CREATE and CREATE, to describe what takes place when a subtransaction is activated. The REQUEST-CREATE is an operation of the transaction's parent, while the actual CREATE takes place at the subtransaction itself. In actual distributed systems such as Argus [LiS], this separation does occur, and the distinction will be important in our results and proofs. Similar remarks hold for the REQUEST-COMMIT and COMMIT operations, which occur at at transaction and its parent, respectively.

We leave the executions of particular transaction automata largely unspecified; the choice of which children to create, and what value to return, will depend on the particular implementation. However, it is convenient to assume that schedules of transaction automata obey certain syntactic constraints. Thus, transaction automata are required to preserve well-formedness, as defined below.

We recursively define *well-formedness* for sequences of operations of a transaction T. Namely, the empty schedule is well-formed. Also, if $\alpha = \alpha'\pi$ is a sequence of operations of T, where π is a single operation, then α is well-formed provided that α' is well-formed, and the following hold:

- If π is CREATE(T), then
  1. there is no CREATE(T) in α'.
- If π is COMMIT(T',v) or ABORT(T') for a child T' of T, then
  1. REQUEST-CREATE(T') appears in α' and
  2. there is no return operation for T' in α'.

- If π is REQUEST-CREATE(T') for a child T' of T, then
  1. there is no REQUEST-CREATE(T') in α'
  2. there is no REQUEST-COMMIT for T in α' and
  3. CREATE(T) appears in α'.
- If π is a REQUEST-COMMIT for T, then
  1. there is no REQUEST-COMMIT for T in α' and
  2. CREATE(T) appears in α'.

These restrictions are very basic; they simply say that a transaction is created at most once, does not receive repeated (or conflicting) notification of the fates of its children, and does not receive information about the fate of any child whose creation it has not requested. Also, a transaction performs output operations neither before it is created nor after it has requested to commit, and a transaction does not request the creation of any given child more than once.

Except for these minimal conditions, there are no restrictions on allowable transaction behavior. For example, the model allows a transaction to request to commit without discovering the fate of all subtransactions whose creation it has requested. Also, a transaction can request creation of new subtransactions at any time, without regard to its state of knowledge about subtransactions whose creation it has previously requested. Particular programming languages may choose to impose additional restrictions on transaction behavior. (An example is Argus, which suspends activity in transactions until subtransactions complete.) However, our results do not require such restrictions.

Basic Objects: Since access transactions model abstract operations on shared data objects, we associate a single I/O automaton with each object, rather than one with each access. The operations of a *basic object* automaton X are the invocation and return operations of the its access transactions:

Input operations:	CREATE(T), for T $\in$ accesses(X)
Output operations:	REQUEST-COMMIT(T,v), for T $\in$ accesses(X) and v $\in V$

Let α be a sequence of operations of basic object X. Then an access T to X is said to be *pending* in α provided that there is a CREATE(T) but no REQUEST-COMMIT for T in α.

It is convenient to require that schedules of basic objects satisfy certain syntactic conditions. Thus, each basic object is required to preserve well-formedness, which is defined recursively as follows.

The empty schedule is well-formed. If $\alpha = \alpha'\pi$ is a sequence of operations of basic object X, where π is a single operation, then α is well-formed provided that α' is well-formed, and the following conditions hold.

- If π is CREATE(T) then
  1. there is no CREATE(T) in α', and
  2. there are no pending accesses in α'.
- If π is a REQUEST-COMMIT for T then
  1. there is no REQUEST-COMMIT for T in α', and
  2. CREATE(T) appears in α'.

That is, the schedules of basic objects are restricted to consist of alternating CREATE and REQUEST-COMMIT operations, starting with a CREATE, and with each (CREATE, REQUEST-COMMIT) pair having the same access transaction, where each access transaction has at most one CREATE.

Serial Scheduler: The serial scheduler is a fully specified automaton. The serial scheduler can choose nondeterministically to abort any transaction T after parent(T) has issued a REQUEST-CREATE(T) operation, as long as T has not actually been created. Thus, the "semantics" of an abort(T) operation are that T was never created. Furthermore, a transaction can only be created if (1) it has not already been created, (2) its parent has requested its creation, and (3) all of its created siblings have returned. In other words, the scheduler runs transactions according to a depth-first traversal of the transaction tree. Finally, the scheduler cannot commit a transaction until all of the transaction's children have returned. The formal definition of the serial scheduler, adapted from [LM,FLMW], is as follows.

The state of the serial scheduler has components create-requested, created, commit-requested, committed, aborted, and returned. Commit-requested is a set of (transaction,value) pairs, and the rest are sets of transaction names. Initially, create-requested $= \{T_0\}$, and the other sets are empty.

The steps of the transition relation for each automaton we define are exactly those triples (s', π, s) satisfying the pre- and postconditions listed, where π is the indicated operation. If a component of s i not mentioned in the postcondition, then it is taken to be the same in s as in s'.

Input operations:	REQUEST-CREATE(T)
	REQUEST-COMMIT(T,v)
Output operations:	CREATE(T)
	COMMIT(T,v)
	ABORT(T)

- REQUEST-CREATE(T)

  Postcondition: create-requested(s) = create-requested(s') $\cup$ {T}

- REQUEST-COMMIT(T,v)

  Postcondition: commit-requested(s) = commit-requested(s') $\cup$ {(T,v)}

- CREATE(T)

  Precondition: T $\in$ create-requested(s') $-$ (created(s') $\cup$ aborted(s'))
  siblings(T) $\cap$ created(s') $\subseteq$ returned(s')

  Postcondition: created(s) = created(s') $\cup$ {T}

- COMMIT(T,v)

Precondition:	(T,v) $\in$ commit-requested(s')
	T $\notin$ returned(s')
	children(T) $\cap$ create-requested(s') $\subseteq$ returned(s')
Postcondition:	committed(s) = committed(s') $\cup$ {(T,v)}
	returned(s) = returned(s') $\cup$ {T}

- ABORT(T)

Precondition:	T $\in$ create-requested(s') $-$ (created(s') $\cup$ aborted(s'))
	siblings(T) $\cap$ created(s') $\subseteq$ returned(s')
Postcondition:	aborted(s) = aborted(s') $\cup$ {T}
	returned(s) = returned(s') $\cup$ {T}

Let S be a serial system, and let σ be a sequence of operations of S. We say that σ is ***well-formed*** iff its projection at every primitive is well-formed. If σ is a schedule of S, then σ is a ***serial schedule***. In [LM], it is shown that all serial schedules are well-formed.

Let S be a serial system, and let γ be an arbitrary sequence of operations. We say that γ ***is serially correct with respect to S for transaction*** T provided that $\gamma|T = \sigma|T$ for some schedule σ of S.

2.3 Model Extensions for Replicated Data Systems

In this section, we add to the model some definitions that are useful for formalizing and understanding replicated data management algorithms.

In order to understand why these particular definitions are useful, it is helpful to keep in mind the general proof strategy we use. As explained in Chapter 1, for each algorithm considered we first construct a serial system in which database items are implemented as multiple replicas, where access to the replicas is controlled by the replication algorithm. Then, we construct a serial system (with the same user transactions) in which each database item is implemented as a single replica. Finally, we prove that each user transaction[2] in the replicated system has the same execution as its corresponding transaction in the non-replicated system.

[2]For each system, we will define formally what is meant by a user transaction in terms of the system type. In general, however, one may think of user transactions as all the non-access transactions that do not model part of the replication algorithm. As a rule, user transactions are those transactions which we do not describe with *fully-specified* automata.

We have already discussed serial systems and provided formal definitions for transactions, accesses, and executions. However, in order to give a more precise meaning to the above description of our proof strategy, we need formal definitions for "database item," "replica," and "corresponding transaction."

Logical Data Items: We refer to database items as "logical data items" to distinguish them from their physical counterparts, the replicas.

A *logical data item* x is a variable, whose type is the tuple $\langle V_x, i_x \rangle$. The set V_x is the domain of possible values for x, and $i_x \in V_x$ is the initial value of x. We require that a special undefined value, *nil*, be an element oı V_x, and that a special place-holder symbol, $\perp$, not be an element of V_x.

Read-write Objects: Each replica is modelled as a fully specified basic object called a *read-write object*, where the domain and initial value depend upon the particular data replication management algorithm and the type of the logical data item. Before we can specify read-write object automata, we require the following definition.

If d and d' are data values from a domain $D \cup \{\perp\}$, then $d \diamond d'$ is defined to be d if d' is $\perp$, and d' otherwise. If $t = \langle d_1, d_2, ... \rangle$ and $t' = \langle d'_1, d'_2, ... \rangle$ are tuples of the same type, then we define $t \diamond t'$ to be the tuple $\langle d_1 \diamond d'_1, d_2 \diamond d'_2, ... \rangle$. This operator allows us to overwrite certain components of a tuple while leaving the other components unchanged.

We now define the concept of a *read-write object with domain D and initial value d*.

The state of a read-write object O with domain D with initial value $d \in D$ consists of two components, active and data. The variable active (initially nil) holds the name of the current access to O. Data holds an element of D (initially d). Every read-write object has a set of accesses, denoted *accesses*(O). Each access T to a read-write object has the attributes *kind*(T) $\in$ {read,write} and *data*(T) $\in D$. When kind(T) = write, data(T) is the data to be written.

Input operations:	CREATE(T), where T $\in$ accesses(O)
Output operations:	REQUEST-COMMIT(T,v), where T $\in$ accesses(O)

- **CREATE(T), for T $\in$ accesses(O)**
  Postcondition: active(s) = T

- **REQUEST-COMMIT(T,v), for kind(T) = read**
  Precondition: active(s') = T
  v = data(s')
  Postcondition: active(s) = nil

- **REQUEST-COMMIT(T,v), for kind(T) = write and data(T) = d**
  Precondition: active(s') = T
  v = nil
  Postcondition: data(s) = data(s') $\diamond$ d
  active(s) = nil

A read-write object accepts read and write accesses. For read accesses, it returns the value in the data component of its state. For write accesses, it applies the write value to its data value using the $\diamond$ operator. For example, if its current data value is $\langle a, b\rangle$ and it processes a write access with data $\langle c, \perp\rangle$, the resulting data component of its state will be $\langle c, b\rangle$.

If T$\in$accesses(O), we say that O(T)=O. That is, we use O(T) to denote the read-write object to which T is an access.

Lemma 2 Read-write objects are basic objects.

Proof: It suffices to show that read-write objects preserve well-formedness of schedules. Let O be a read-write object. Let $\alpha = \alpha'\pi$ be a schedule of O, where π = REQUEST-COMMIT(T,v), and assume that α' is well-formed. We must show that: (1) CREATE(T) occurs in α', and (2) no REQUEST-COMMIT for T occurs in α'. These properties are guaranteed by the use of the variable active. A precondition for REQUEST-COMMIT for T is that active = T. Since only a CREATE(T) can cause active to equal T, part (1) holds. Assume, for contradiction, that a REQUEST-COMMIT for T occurs in α'. By part (1), the REQUEST-COMMIT for T must occur after a CREATE(T). A postcondition of REQUEST-COMMIT for T is that active = nil. Therefore, the state of O after α' has active $\neq$ T, because well-formedness implies that α' contains at most one CREATE(T) operation.

However, if active $\neq$ T, then π (REQUEST-COMMIT for T) is not enabled in O after α'. But $\alpha'\pi$ is a schedule of O, giving us a contradiction. Thus, part (2) holds. ■

Extensions of Systems: We want to define formally the notion of "corresponding transactions" so that we can be precise in our comparisons of each pair of replicated and non-replicated systems. That is, for certain pairs of systems, we would like a function that maps each transaction of one system to some transaction in the other system. In order for this function to be well-defined, we must impose certain restrictions on the system types of the two systems.

Let S' and S be two systems with system types Σ' and Σ, respectively. System type Σ' *is an extension of* system type Σ if the transaction tree of Σ is a subgraph of the transaction tree of Σ' and both trees have the same root. If Σ' is an extension of Σ, then we say that system S' *is an extension of* system S.

If system S' is an extension of system S, relating the transactions in the two systems is easy. We define function $\mathcal{F}_{SS'} : \mathcal{T}_S \rightarrow \mathcal{T}_{S'}$ to map transactions in S to their same-named transactions in S'. The inverse, $\mathcal{F}_{S'S}$, is a partial function unless S and S' have the same transaction tree.

Configurations: As a final addition to the model, we introduce the following general definitions, which are central to the algorithms we study.

Let S be any arbitrary set, and let Q be the power set 2^S. We define *configurations*(S) to be the set of all pairs of the form $\langle r, w\rangle$, where $r, w \subseteq Q$. (We sometimes refer to r and w as sets of read-quorums and write-quorums, respectively.) The set *legal*(S) is defined to be the set of all elements $\langle r, w\rangle$ of configurations(S) such that every element of r has a non-empty intersection with every element of w.

We say that every element of configurations(S) is a *configuration* of S, and that every element of legal(S) is a *legal configuration* of S.

Notation: We let N denote the set of non-negative integers (i.e., {0,1,2,...}).

Chapter 3

Fixed Quorum Consensus

In this chapter, we formalize and prove the correctness of a generalized version of Gifford's algorithm without reconfiguration, as described in the introduction. In Section 3.1, we define system B, a ***replicated serial system*** that uses the fixed quorum consensus algorithm to manage replicas, and prove some properties of its schedules. Then, in Section 3.2, we define a corresponding ***non-replicated serial system,*** named system A. We prove the correctness of the fixed quorum consensus algorithm in Section 3.3 by showing that system B simulates system A in a strong sense. Finally, in Section 3.4, we show that non-serial replicated systems are correct.

3.1 Replicated Serial System

The replicated serial system defined in this section is an ordinary serial system in which certain logical data items are replicated. That is, they are implemented as several basic objects (replicas), rather than just one. We impose a restriction on the transaction tree so that all accesses to the replicas are the children of transaction manager automata (TMs), which we define explicitly. The TMs model the Quorum Consensus algorithm itself. We model the read and write operations of the algorithm by providing two kinds of TMs, read-TMs and write-TMs. We place no restrictions on the remaining automata, except that they preserve well-formedness. The system is formally defined as follows.

Fix I, a set of logical data items. We define system B to be a serial system of type

$\langle \mathcal{T}, \text{parent}, \mathcal{O}, V \rangle$. For each element x of I, we define:

- $dm(x)$, a subset of $\mathcal{O}$,
- $acc(x)$, a subset of the accesses in $\mathcal{T}$
- $tm_r(x)$ and $tm_w(x)$, disjoint subsets of the non-accesses in $\mathcal{T}$,
- $config(x)$, a legal configuration of $dm(x)$.

Let $tm(x) = tm_r(x) \cup tm_w(x)$. We require that $acc(x)$ is exactly the set of all accesses to objects in $dm(x)$. In our replicated serial system, the replicas for x will be associated with the members of $dm(x)$, and the logical accesses to x will be managed by automata associated with the members of $tm(x)$. Since we want all accesses to replicas for x to be controlled by the replication algorithm, we require that T$\in acc(x)$ iff parent(T)$\in tm(x)$. Finally, for all pairs $x, y \in I$, we require that $dm(x) \cap dm(y) = \emptyset$.

We define the ***user transactions*** in B to be the set of non-access transactions in $\mathcal{T}$ that are not in $tm(x)$ for all $x \in I$. We refer to accesses in $acc(x)$ for all $x \in I$ as ***replica accesses***, and to the remaining accesses in $\mathcal{T}$ as ***non-replica accesses***.

Figure 3.1 provides an example of a possible transaction tree for system B.

In system B, each member of $dm(x)$ has a corresponding data manager automaton (DM) for x, each member of $tm_r(x)$ has an associated read-TM automaton for x, and each member of $tm_w(x)$ has an associated write-TM automaton for x. From the restrictions on the system type, then, the members of $acc(x)$ are the accesses to the DMs for x. Furthermore, the accesses to DMs for x are exactly the children of the TMs for x. DMs and TMs for x are described below.

Data Managers: The set of data managers for logical data item x models the set of physical replicas of x. Each DM is a read-write object that keeps a version-number and a value for x. The formal definition follows.

If x is a logical data item, a *DM* for x is a read-write object over domain $D_x = N \times V_x$ with initial data $\langle 0, i_x \rangle$. We refer to each member of D_x as a $\langle$version-number,value$\rangle$ pair. (For

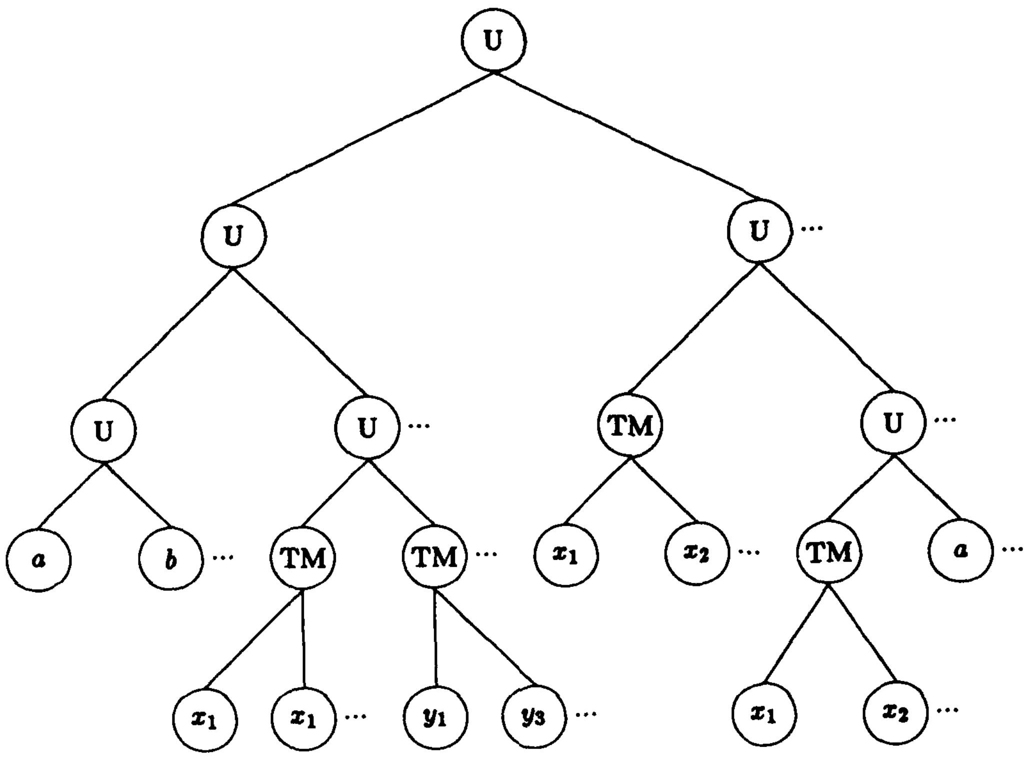


Figure 3.1: A possible transaction tree for system B. Transactions are labeled as follows: U = user transaction; TM = transaction manager; a, b = non-replica accesses; x_1 = replica access to replica 1 of logical data item x, etc.

v$\in D_x$, we use the record notation v.version-number and v.value to refer to the components of v.)

Lemma 3 DMs are basic objects.

Proof: Immediate from Lemma 2. ■

Recall that we have restricted the system type of B so that accesses to DMs for x are invoked only by TMs for x. We now define read-TMs and write-TMs for x.

Read TMs: Let x be a logical data item in I. The purpose of a read-TM for x is to perform a logical read access to x. A read-TM for x invokes read accesses to multiple DMs for x. It then returns the "current" value of x, which it calculates from the information returned by the read accesses. In Lemma 8, we show that read-TMs in system B do, in fact, return the proper value of x. That is, a read-TM returns the value that would be expected, given the sequence of logical write accesses to x that precedes its invocation.

A read-TM T for x has state components awake, data, requested, and read, where awake is a boolean value, data is a value in the domain D_x, requested is a subset of $acc(x)$, and read is a subset of $dm(x)$. Initially, data is $\langle 0, i_x \rangle$, awake is false, and requested and read are both empty.

Note: Whenever an undefined variable (for example, q in the REQUEST-COMMIT operation of the following automaton) appears in the pre- and/or postconditions for an operation, then that variable has an implicit existential quantifier (i.e., there exists a q such that...).

Input operations:	CREATE(T)
	COMMIT(T',v), where T' $\in$ children(T) and v $\in D_x$
	ABORT(T'), where T' $\in$ children(T)
Output operations:	REQUEST-CREATE(T'), where T' $\in$ children(T)
	REQUEST-COMMIT(T,v), where v $\in D_x$

- CREATE(T)
  Postcondition: awake(s) = true

- REQUEST-CREATE(T'), where kind(T') = read
  Precondition: awake(s') = true
  T' $\notin$ requested(s')
  Postcondition: requested(s) = requested(s') $\cup$ {T'}

- COMMIT(T',v)
  Postcondition: read(s) = read(s') $\cup$ {O(T')}
  if v.version-number > data(s').version-number then data(s) = v

- ABORT(T')
  Postcondition: (no change)

- REQUEST-COMMIT(T,v)
  Precondition: awake(s') = true
  $q \in config(x).r$
  $q \subseteq$ read(s')
  v = data(s').value
  Postcondition: awake(s) = false

A read-TM collects data from some number of DMs for x, always keeping the data from the DM with the highest version number seen so far. When a read-quorum of DMs has been seen, the read-TM may request to commit and return its data.

It is interesting to note the extensive use of nondeterminism in this algorithm. For example, *the read-TM does not set out* to access any particular read-quorum in the configuration. Rather, the read-TM simply invokes any number of accesses to any of the DMs until it happens to notice that COMMIT operations have been received from some read-quorum of DMs. Also, since it is not necessary for correctness (as opposed to efficiency) for the read-TM to remember which of its children have aborted, the ABORT(T') operation has no postconditions.

The nondeterminism allows for greater generality of our results. However, one would not want to implement read-TMs this loosely in a real system. For the sake of efficiency, one would want to limit the number of accesses invoked by a read-TM. For example, one would want the read-TM to invoke accesses with some particular read-quorum in mind. Also, one would not want the read-TM to invoke an access to a DM from which it has already received information. Similarly, one might not want the read-TM to invoke an access to a DM if several previous accesses to that DM have aborted. The important point, however, is that all of our results apply even if such heuristics are added. Our proofs depend only upon the fact that all operations performed satisfy the preconditions and postconditions we define.

Write TMs: Let x be a logical data item in I. The purpose of a write-TM for x is to perform a logical write access to x. The formal description of a write-TM automaton follows.

A write-TM T for x has state components awake, data, read-requested, write-requested, read and written, where awake is a boolean variable, data is an element of D_x, read-requested and write-requested are subsets of $acc(x)$, and read and written are subsets of $dm(x)$. Initially, data = $\langle 0, i_x \rangle$, awake is false, and the sets are empty. Every write-TM T for x has an associated value $value$(T) $\in V_x$, the value to be written to the logical data item.

Input operations:	CREATE(T)
	COMMIT(T',v), where T' $\in$ children(T) and v$\in D_x$
	ABORT(T'), where T' $\in$ children(T)
Output operations:	REQUEST-CREATE(T'), where T' $\in$ children(T)
	REQUEST-COMMIT(T,v), where v = nil

- CREATE(T)

  Postcondition: awake(s) = true

- REQUEST-CREATE(T'), where kind(T') = read

```
Precondition:   awake(s') = true
                T' $\notin$ read-requested(s')
Postcondition:  read-requested(s) = read-requested(s') $\cup$ {T'}
```

- COMMIT(T',v), where kind(T') = read

```
Postcondition:  if write-requested(s') = {} then
                    read(s) = read(s') $\cup$ {O(T')}
                    if v.version-number > data(s').version-number then
                        data(s).version-number = v.version-number
```

- REQUEST-CREATE(T'), where kind(T') = write and data(T') = d

```
Precondition:   awake(s') = true
                $q \in config(x).r$
                $q \subseteq$ read(s')
                d = $\langle$data(s').version-number+1,value(T)$\rangle$
                T' $\notin$ write-requested(s')
Postcondition:  write-requested(s) = write-requested(s') $\cup$ {T'}
```

- COMMIT(T',v), where kind(T') = write

  Postcondition: written(s) = written(s') $\cup$ {O(T')}

- ABORT(T')

  Postcondition: (no change)

- **REQUEST-COMMIT(T,v)**
  Precondition: **awake = true**
  v = nil
  $q \in config(x).w$
  $q \subseteq$ **written(s')**
  Postcondition: **awake = false**

A write-TM invokes read accesses to some number of DMs for x, keeping track of the highest version number returned. Once information from a read-quorum of DMs has been collected, the write-TM may begin invoking write accesses. (See the REQUEST-CREATE(T') operation.) The version-number of each write access invoked is one greater than the version-number in the data component of the write-TM's state, and the value of each write access invoked is value(T). Once COMMIT operations have been received from a write-quorum of DMs, the write-TM may request to commit.

It is possible that some read accesses to the DMs may not commit until after the write-TM has already invoked one or more write accesses. Thus, some read accesses may actually return the data that was written to the DMs on behalf of the write-TM itself. Therefore, in order to prevent the write-TM from seeing the data it wrote and incorrectly increasing its version-number, the COMMIT operation for read accesses is defined so that the state of the write-TM is modified only if no write accesses have been invoked.

Our discussion of the nondeterminism in read-TMs also applies to write-TMs, as well as to all other automata we define.

Lemma 4 TMs are transactions.

Proof: Let T be one of these automata. It suffices to show that T preserves well-formedness. Let $\alpha = \alpha'\pi$ be a schedule of T where π is an output operation, and assume that α' is well-formed. We need to show that: (1) CREATE(T) occurs in α', (2) no REQUEST-COMMIT operation for T occurs in α', and (3) if π is a REQUEST-CREATE(T') operation, then no REQUEST-CREATE(T') occurs in α'. By the definition of T, no output operation can be issued if awake = false and only the CREATE operation can set awake to true. Therefore, part (1) is true. Only the REQUEST-COMMIT operation can set awake to false and by definition of well-formed schedule, α' can contain at most one CREATE operation

(Once awake becomes false, it remains false forever.) Therefore, part (2) holds. Whenever a REQUEST-CREATE(T') operation is performed, that fact is remembered permanently in the state of T. A precondition for REQUEST-CREATE(T') is that T' has not previously been created by T. Since T= parent(T'), only T may issue a REQUEST-CREATE(T') operation. Therefore, part (3) holds. ∎

Lemma 5 Schedules of system B are well-formed.

Proof: By Lemmas 3 and 4, DMs are basic objects and TMs are transactions. Therefore, system B is a serial system. In [LM], it is proved that all schedules of serial systems are well-formed. ∎

The following definitions are useful for describing the logical accesses to the logical data items in system B and for setting up inductive arguments about these logical accesses.

Access sequence: This definition formalizes the intuitive notion of a sequence of logical accesses to x.

Let β be a sequence of operations of system B, and let x be a logical data item in I. Then the *access sequence* of x in β, denoted $access(x, \beta)$, is defined to be the subsequence of β containing the CREATE and REQUEST-COMMIT operations for the members of $tm(x)$.

Logical state: The following definition formalizes the intuitive notion of the "current state" of a logical data item, the expected return value of a logical read.

Let β be a sequence of operations of system B, and let x be a logical data item in I. The *logical state of x after β*, denoted $logical\text{-}state(x, \beta)$, is defined to be either $value$(T) if REQUEST-COMMIT(T,v) is the last REQUEST-COMMIT operation for a write-TM in access(x, β), or i_x if no REQUEST-COMMIT operation for a write-TM occurs in access(x, β).

Current version number: Let β be a sequence of operations of system B, and let x be a logical data item in I. Let $last(x, \beta)$ denote the subset of $acc(x)$ such that for each member T of last(x, β), REQUEST-COMMIT for T is the last REQUEST-COMMIT operation for a write access to O(T) in β.[1] The ***current version number of x after β***, denoted ***current-vn***(x, β), is defined as follows. If last(x, β) is non-empty, then current-vn(x, β) is the maximum over all T$\in$last(x, β) of data(T).version-number. Otherwise, current-vn(x, β) = 0.

Lemma 6 If β is a schedule of B and x is a logical data item in I, then access(x, β) begins with a CREATE operation for some TM in $tm(x)$ and continues alternately with REQUEST-COMMIT and CREATE operations for TMs in $tm(x)$ such that each REQUEST-COMMIT for T is preceded immediately by a CREATE(T) operation.

Proof: By definition, access(x, β) contains only CREATE and REQUEST-COMMIT operations for TMs in $tm(x)$. By Lemma 5, β is a well-formed schedule, so each REQUEST-COMMIT for T must be preceded by a CREATE(T) operation. Finally, since β is a serial schedule, all operations for a given transaction must be contiguous. ■

Lemma 7 Let x be a logical data item, and let β be a schedule of B. Then the following property holds after β: The highest version number among the states of all DMs in $dm(x)$ is current-vn(x, β).

Proof: Since DMs are read-write objects, the only operation that can change the version-number in the state of a DM O for x is a REQUEST-COMMIT for T operation, where O(T) = O and T is a write access. More specifically, the version-number in the state of a DM O after β is data(T).version-number, where REQUEST-COMMIT for T is the last such REQUEST-COMMIT in β. In the definition of current-vn(x, β), the set last(x, β) contains the last write access for each DM in $dm(x)$ that has a REQUEST-COMMIT for a write access in β. Therefore, the maximum over all T$\in$last(x, β) of data(T).version-number is the highest version number among the states of all DMs in $dm(x)$ after β. This maximum is exactly the definition of current-vn(x, β). ■

The following lemma is the key to the proof of Theorem 10. Condition 1 is only needed for carrying through the inductive argument. The important part of the lemma is Condition 2, which tells us that each read-TM returns the value expected as dictated by the

[1]Note that the cardinality of last(x, β) equals the number of DMs for x having write accesses that request to commit in β.

previous logical write operations. That is, each read-TM returns the logical-state of the data item. Because the system is serial, we are able to carry out a simple inductive proof using standard assertional techniques. In the proof of this lemma, as well as the proofs of the remaining lemmas and theorems, we formally consider all details except the preconditions and postconditions for the operations of the basic objects; because their behavior is so simple, these operations receive only informal treatment.

Lemma 8 Let x be a logical data item in I. Let β be a schedule of B such that access(x, β) is of even length.

1. The following properties hold after β:
   (a) There exists a write-quorum $q \in config(x).w$ such that for all DMs $O \in q$, if d is the data component of O, then d.version-number = current-vn(x, β).
   (b) For all DMs $O \in dm(x)$, if d is the data component of O, then d.version-number = current-vn(x, β) implies that d.value = logical-state(x, β).
2. If β ends in REQUEST-COMMIT(T,v) with T$\in tm_r(x)$, then v = logical-state(x, β)

Proof: By induction on the length of β.

Base case. Let β be the empty schedule. By definition, current-vn(x, β) = 0 and logical-state(x, β) = i_x. Initially, all DMs in $dm(x)$ have version-number = 0 and value = i_x by the definition of a DM. Therefore, the states after β of all the DMs in every $q \in config(x).w$ have version-number = current-vn(x, β) and value = logical-state(x, β). Thus, part 1 holds. Since β is empty, it does not end in a REQUEST-COMMIT operation of a read-TM for x. So, part 2 holds vacuously.

Induction. Let $\beta = \beta' \tau$, where access(x, τ) begins with the last CREATE operation in access(x, β). Assume that the Lemma holds for β'. By Lemma 6 and the fact that access(x, β) is of even length, access(x, τ) = (CREATE(T$_f$), REQUEST-COMMIT(T$_f$,v$_f$)) for some T$_f \in tm(x)$ and v$_f \in V_x$. We note the following facts about T$_f$.

Fact 1 All accesses in τ to DMs in $dm(x)$ are descendants of T$_f$.

Proof. Since β is a serial schedule, T$_f$ is the only TM in $tm(x)$ whose descendants have operations in r. Furthermore, the system type of B is constrained so that all accesses to DMs in $dm(x)$ are children of TMs in $tm(x)$.

Fact 2: Let s be the state of T_f after any prefix of β. If read(s) is non-empty, then data(s).version-number and data(s).value contain the highest version-number and associated value among the states of the DMs in read(s) after β'.

Proof: By definition, a DM O is added to the read component of T_f only as the result of a COMMIT(T',v') operation, where parent(T') = T_f, T' is a read access, O(T) = O, and T_f has not invoked any prior REQUEST-CREATEs for write accesses[2]. Since β is well-formed, all such COMMIT(T',v') operations must occur in r. By Fact 1, all accesses to O that take place in r are children of T_f. Since T_f invokes no write accesses prior to the COMMIT(T',v') operation, the data components of the DMs in $dm(x)$ for the COMMIT(T',v') operation are the same as after β'. Therefore, v' is the data component of the state of O after β'. By definition, T_f retains the maximum version-number (and its associated value) among all the return values of COMMIT for T' operations that result in O(T') being added to the read component.

Fact 3: Let s be the state of T_f after any prefix of β. If read(s) contains some read quorum $r \in$ *config(x).r*, then data(s).version-number = current-vn(x, β') and data(s).value = logical-state(x, β').

Proof: By the induction hypothesis, there exists some write-quorum $w \in config(x).w$ such that the states of all DMs in w after β' have version-number = current-vn(x, β'), and every DM with version-number current-vn(x, β') has value = logical-state(x, β'). By Lemma 7, current-vn(x, β') is the highest version number among all DMs in $dm(x)$. Since $config(x)$ is a legal configuration of $dm(x)$, r and w must have a non-empty intersection. So, read(s) must contain at least one DM in w. Therefore, by Fact 2, data(s).version-number = current-vn(x, β') and data(s).value = logical-state(x, β').

From Fact 1, we know that all accesses to DMs for x in r are children of T_f. Therefore, in order to prove that the induction hypothesis holds for β, we merely need to demonstrate that T_f preserves the properties stated. There are two possibilities for T_f.

[2]This final condition is trivially true when T_f is a read-TM.

- If T_f is a read-TM, then logical-state(x,β) = logical-state(x,β') by definition. Also, since T_f invokes only read accesses, the version-number and value components of the states of the DMs in $dm(x)$ after β are the same as after β', and current-vn(x,β) = current-vn(x,β'). Therefore, part 1 of the Lemma holds for β.

  Let s_f be the state of T_f when T_f issues its REQUEST-COMMIT operation. The preconditions for REQUEST-COMMIT require that read(s_f) contain some read-quorum $r \in config(x).r$. Therefore, by Fact 3, data(s_f).value = logical-state(x,β'), which equals logical-state(x,β). By definition, v_f = data(s_f).value, so part 2 of the Lemma holds for β.

- If T_f is a write-TM, then logical-state(x,β) = value(T_f) by definition. We note the following fact about T_f:

  Fact 4: For all write accesses T' invoked by T_f, data(T') = ⟨current-vn(x,β')+1, value(T_f)⟩.

  Proof: Let s_w be the state of T_f when it issues REQUEST-CREATE(T'). By definition, data(T') = ⟨data(s_w).version-number+1,value(T_f)⟩. The precondition for the REQUEST-CREATE(T') operation requires that read(s_w) contain some read quorum $q \in config(x).r$. Therefore, by Fact 3, data(s_w).version-number = current-vn(x,β').

  Let s_f be the state of T_f when T_f issues its REQUEST-COMMIT operation. The preconditions for REQUEST-COMMIT require that written(s_f) contain some write-quorum $w \in config(x).w$. Furthermore, no DM is added to the written component of the state of T_f unless a write access to that DM has committed to T_f. So, τ must contain a REQUEST-COMMIT operation for a write access to each DM in w. After a COMMIT of a write access T' to a DM, the data component of that DM is equal to data(T'). Therefore, by Fact 4, the states after β of all the DMs in w must have value = value(T_f) and version-number = current-vn(x,β')+1. (By Fact 1, T_f is the only transaction that issues write accesses to DMs in $dm(x)$ in τ.) By Lemma 7, current-vn(x,β') is the highest version-number among the states of DMs in $dm(x)$ after β'. Since every write access in τ to DMs in $dm(x)$ has version-number = current-vn(x,β')+1, we know that this is the highest version-number among DMs in $dm(x)$ after β. That is, current-vn(x,β')+1 = current-vn(x,β). Therefore, since value(T_f) = logical-state(x,β), part 1 of the Lemma holds. Since T_f is not a read-TM, β does not end with a REQUEST-COMMIT of a read-TM for x, so part 2 holds vacuously.

Thus, the Lemma holds in both cases. ∎

3.2 Non-replicated Serial System

As the basis of our correctness condition, we define non-replicated serial system A of type ($\mathcal{T}_A$,parent$_A$,$\mathcal{O}_A$,V_A) in terms of replicated serial system B of type ($\mathcal{T}_B$,parent$_B$,$\mathcal{O}_B$,V_B).[3] System A is identical to System B, except that logical accesses to objects in $\mathcal{I}$ (which are implemented as TMs in system B) are implemented as accesses in system A, and the logical data items in $\mathcal{I}$ (which are implemented as collections of DMs in system B) are implemented as single read-write objects in system A. These changes are reflected in the system type, which is formally defined as follows:

- $\mathcal{T}_A = \mathcal{T}_B - \left(\bigcup_{x\in \mathcal{I}} acc(x)\right)$
- parent$_A$ = parent$_B$ restricted to $\mathcal{T}_A$
- $\mathcal{O}_A = \mathcal{O}_B - \left(\bigcup_{x\in \mathcal{I}} dm(x)\right) \cup \{tm(x) | x \in \mathcal{I}\}$
- $V_A = V_B$

Informally, to construct the type of system A from that of system B, we first remove from $\mathcal{T}$ all the accesses to the DMs for objects in $\mathcal{I}$. As a result, all the TMs for objects in $\mathcal{I}$ become leaves in $\mathcal{T}$ and are therefore accesses. Next, we remove from $\mathcal{O}$ all the DMs for objects in $\mathcal{I}$. Also, we partition all the accesses that were formerly TMs according to their logical data item. Each class of this partition is a new object in $\mathcal{O}$. Thus, each logical data item is implemented by a single object.

[3]We introduce the subscripts to distinguish the components of A from the components of B.

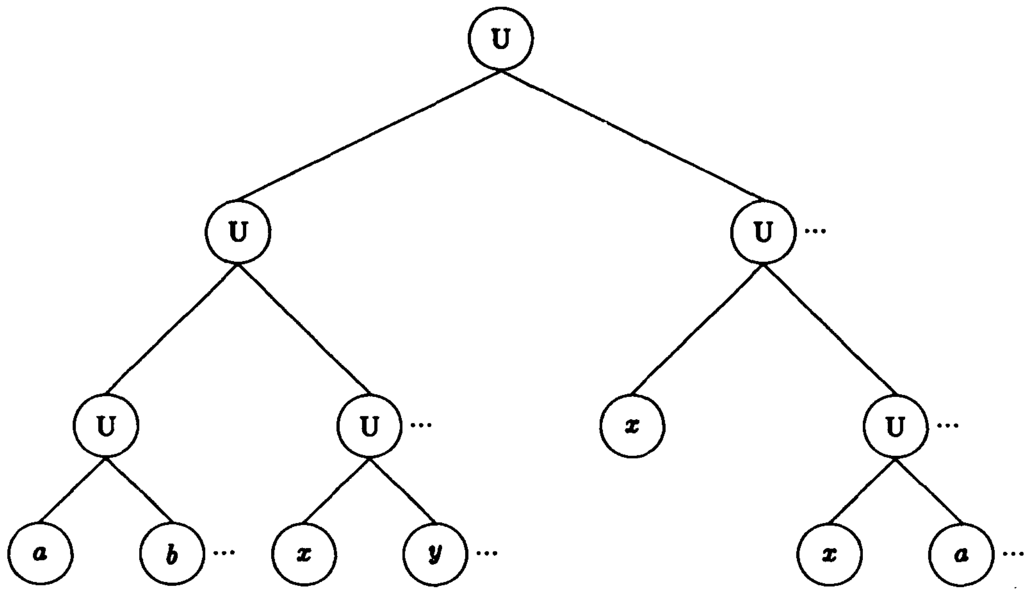


Figure 3.2: The transaction tree for system A that corresponds to the transaction tree for B shown in Figure 3.1. Transactions are labeled as follows:
U = user transaction; a, b, x, y = accesses.

Figure 3.2 illustrates the transaction tree for system A that corresponds to the transaction tree for system B given in Figure 3.1.

We would like to relate transactions in system B to those in system A. Recall that the function $\mathcal{F}_{AB}$ is well-defined, provided that system B is an extension of system A. Thus, we prove the following lemma.

Lemma 9 System B is an extension of system A.

Proof: We need to show that $\mathcal{T}_A \subseteq \mathcal{T}_B$ and that $\mathcal{T}_A$ has the same root as $\mathcal{T}_B$. Since $\mathcal{T}_A = \mathcal{T}_B - (\bigcup_{x \in \mathcal{I}} acc(x))$, we know that $\mathcal{T}_A \subseteq \mathcal{T}_B$. Furthermore, $\mathcal{T}_A$ and $\mathcal{T}_B$ must have the same root, unless the root of $\mathcal{T}_B$ is in $acc(x)$ for some $x \in \mathcal{I}$. However, every member of $acc(x)$ is a child of some member of $tm(x)$, so no access in $acc(x)$ for any $x \in \mathcal{I}$ could be the root of $\mathcal{T}_B$. ■

We define ***user transactions*** in system A to be all non-access transactions in $\mathcal{T}_A$. We note that T is a user transaction in system B iff $\mathcal{F}_{BA}(\mathrm{T})$ is a user transaction in system A. This is because if T is a TM in system B, then $\mathcal{F}_{BA}(\mathrm{T})$ is an access transaction.

Transactions and objects in system A have the same corresponding automata as in system B, except that for all $x \in I$, the following hold:

1. The object corresponding to $tm(x)$ is modelled as a read-write object O over domain V_x with initial value i_x. (We refer to this particular read-write object as $\mathrm{O}(x)$.)
2. For each transaction T$\in tm(x)$, $\mathcal{F}_{BA}(\mathrm{T})$ is an access to $\mathrm{O}(x)$ such that
   (a) if T is a read-TM, then $\mathcal{F}_{BA}(\mathrm{T})$ is a read access, and
   (b) if T is a write-TM, then $\mathcal{F}_{BA}(\mathrm{T})$ is a write access with $\mathrm{data}(\mathcal{F}_{BA}(\mathrm{T})) = \mathrm{value}(\mathrm{T})$.

3.3 Correctness

In this section, we prove that system B is correct by showing that user transactions cannot distinguish between replicated serial system B and non-replicated serial system A.

Theorem 10 Let β be a schedule of replicated serial system B. There exists a schedule α of non-replicated serial system A such that the following two conditions hold.

1. For all objects O in system B that are not in $dm(x)$ for any x, $\alpha|\mathrm{O} = \beta|\mathrm{O}$.
2. For all user transactions T in system B, $\alpha|\mathcal{F}_{BA}(\mathrm{T}) = \beta|\mathrm{T}$.

Proof: We construct α by removing from β all the REQUEST-CREATE(T), CREATE(T), REQUEST-COMMIT(T,v), COMMIT(T,v), and ABORT(T) operations for all transactions T in $acc(x)$ for all $x \in I$. Clearly, the two conditions hold. What needs to be proved is that α is a schedule of A. We proceed by induction on the length of β.

Base case: Suppose β is empty. Then α is also empty and is therefore a schedule of A.

Induction: Let $\beta = \beta'\pi_\beta$, where the claim holds for β'. Let $\alpha = \alpha'\pi_\alpha$, where α' is the schedule of A corresponding to β'. There are five cases for π_β.

1. **Invocations, Operations, and Returns of Replica Accesses:** If π_β is a REQUEST-CREATE(T), CREATE(T), REQUEST-COMMIT(T,v), COMMIT(T,v), or ABORT(T) operation where T is in $acc(x)$ and $x \in I$, then by the construction π_α is empty. Therefore, α is the same as α', which is a schedule of A.

2. **REQUEST-COMMITs for Non-Replica Accesses:** If π_β is a REQUEST-COMMIT for a non-replica access T, then by the construction $\pi_\alpha = \pi_\beta$. By Part 1 of the induction hypothesis, $\alpha'|\mathrm{O} = \beta'|\mathrm{O}$. Since O is modelled by the same automaton in both systems A and B, the states of O after α' and after β' are the same. Furthermore, $\mathcal{F}_{BA}(\mathrm{T})$ is modelled by the same automaton as T. Therefore, since the preconditions for π_β are satisfied in the state of T after β', they must also be satisfied in the state of $\mathcal{F}_{BA}(\mathrm{T})$ after α'. Therefore $\alpha|\mathcal{F}_{BA}(\mathrm{T})$ is a schedule of $\mathcal{F}_{BA}(\mathrm{T})$. So, by the Composition Lemma (Lemma 1), α is a schedule of A.

3. **Output Operations of User Transactions:** If π_β is an output operation of some user transaction T, then by the construction $\pi_\alpha = \pi_\beta$. By Part 2 of the induction hypothesis, $\alpha'|\mathcal{F}_{BA}(\mathrm{T}) = \beta'|(\mathrm{T})$. Furthermore, $\mathcal{F}_{BA}(\mathrm{T})$ is modelled by the same automaton as T. Therefore, since the preconditions for π_β are satisfied in the state of T after β', they must also be satisfied in the state of $\mathcal{F}_{BA}(\mathrm{T})$ after α'. Therefore, $\alpha|\mathcal{F}_{BA}(\mathrm{T})$ is a schedule of $\mathcal{F}_{BA}(\mathrm{T})$. So, by the Composition Lemma, α is a schedule of A.

4. **Output Operations of TMs** (except those already covered by Case 1): If π_β is a REQUEST-COMMIT(T,v), where T$\in tm(x)$ for some $x \in I$, then by the construction $\pi_\alpha = \pi_\beta$. By the definition of system A, $\mathcal{F}_{BA}(\mathrm{T})$ is an access to a read-write object. The only precondition for a REQUEST-COMMIT of T, then, is that T has been created. By the construction and the fact that β is a well-formed schedule, CREATE(T) occurs in α'. Therefore, the precondition for REQUEST-COMMIT(T,v') is satisfied in A for some v'.

   If T is a write-TM, then $v = v' = nil$. We need to show that $v = v'$ if T is a read-TM. By Lemma 8, we know that $v =$ logical-state(x, β'). By definition of a read-write object, v' is the value in the state of O(x) after α'. We observe that, by the construction, $\alpha'|\mathrm{O}(x) = \mathrm{access}(x, \beta')$. So, by definition of system A, the last write access in α' to O(x) has the same value as the last write-TM in β'. Hence, the value in the state of O(x) after α' is logical-state(x, β'). Therefore, $v = v'$. (If there is no write-TM in access(x, β'), then there are no write accesses to O(x) in α'. In this case, the value in the state of O(x) after α' is i_x, which is logical-state(x, β').)

5. **Output Operations of the Scheduler** (except those already covered by Case 1): If π_β is a CREATE(T), a COMMIT for T, or an ABORT(T), where T is a user transaction, T is a non-replica access, or T $\in tm(x)$ for some $x \in I$, then by the construction $\pi_\alpha = \pi_\beta$.

   If π_β is a CREATE(T) or ABORT(T), then the preconditions for π_α are (1) there must be a REQUEST-CREATE(T) in α' but no CREATE(T) or ABORT(T) in α', and (2) all siblings of $\mathcal{T}_{BA}$(T) with creates in α' must have returned (committed or aborted) in α'. Since β is a well-formed schedule, REQUEST-CREATE(T) is in β', and, by the construction, is in α' as well. Similarly, since no CREATE(T) or ABORT(T) can occur in β', none can occur in α' either. Therefore, precondition (1) is satisfied. By the construction, all commits and aborts in α' of siblings of $\mathcal{T}_{BA}$(T) must also appear in β'. So, since β is well-formed, precondition (2) must also be satisfied.

   If π_β is a COMMIT(T,v), then the preconditions for π_α are (1) a REQUEST-COMMIT(T,v) must occur in α', (2) $\mathcal{T}_{BA}$(T) cannot have a COMMIT or ABORT in α', and (3) any children invoked by $\mathcal{T}_{BA}$(T) must have returned in α'. Using the same argument as above, by the construction and the fact that β is well-formed, preconditions (1) and (2) must be satisfied. If T is a non-replica access or T $\in tm(x)$ for some $x \in I$, then $\mathcal{T}_{BA}$(T) cannot have any children in A. If T is a user transaction, then all return operations of the children of T in β' are, by the construction, included in α'. Therefore, since β is well-formed, precondition (3) must be satisfied.

In all cases, α is a schedule of A. ■

3.4 Concurrent Replicated Systems

So far, we have been able to deal exclusively with serial systems in order to simplify our reasoning. We now complete the correctness proof by showing that non-serial replicated systems are correct. Recall the definition of serial correctness: Let S be a serial system, and let γ be an arbitrary sequence of operations. We say that γ ***is serially correct with respect to S for transaction*** T provided that $\gamma|T = \sigma|T$ for some schedule σ of S.

With the following theorem, we show that given a correct concurrency control algorithm, combining that algorithm with our replication algorithm yields a correct system. This theorem allows us to achieve a complete separation of the issues of concurrency control and recovery from the issues of replication. In other words, one may prove a concurrency control algorithm correct, then separately prove a replication algorithm correct for serial systems, and finally apply this theorem to show that the (combined) concurrent replicated system is correct. The modularity of this proof method permits us to ignore all the complicated interactions of the two algorithms that one would need to consider in a direct proof that the concurrent replicated system simulates a non-replicated serial system.

Theorem 11 Let C be any system that has the same type as system B, and let the set of user transactions in C be the same as in B. Assume that all schedules γ of C are serially correct with respect to serial system B for all non-orphan[4] non-access transactions. Then all schedules γ of C are serially correct with respect to system A for all non-orphan user transactions.

Proof: An immediate consequence of Theorem 10. ∎

So, any concurrency control algorithm that provides serializability at the level of the copies may be combined with the Fixed Quorum Consensus replica management algorithm to produce a correct system. Interesting concurrency control algorithms that satisfy this condition include Reed's multi-version timestamp concurrency control algorithm [R] and Moss' two phase locking algorithm with separate read and write locks [Mo]. (See also the correctness proof given by Fekete et al. [FLMW].)

[4]A a transaction T is an orphan in γ if ABORT(T') occurs in γ for some ancestor T' of T.

Chapter 4

Reconfigurable Quorum Consensus

We now extend the results of Chapter 3 to systems that permit reconfiguration. That is, we permit the read- and write-quorums to change dynamically, rather than fixing them for the entire execution. This flexibility is important for coping with site and link failures in practical systems. For example, if some DMs are down, we may want to change the quorums so that logical accesses can be processed in spite of the failures.

We redefine systems A and B and present proofs analogous to those for the fixed configuration systems. In doing so, some interesting new considerations arise: As before, the logical accesses are described in terms of read- and write-TMs. However, we also need a new kind of TM, called a reconfigure-TM, to effect changes in the quorums. We would like the reconfigure-TMs to be modelled as transactions for the sake of uniformity, and to be positioned in the tree as children of the user transactions in order to model the correct atomicity requirements. For instance, if T and T' are TMs for x that are invoked by the same user transaction, we would like to permit reconfiguration of x to take place between the COMMIT of T and the CREATE of T'. However, the reconfigure-TMs are special in that their invocations and returns are not to be controlled, or even seen, by the user transactions. Rather, they are intended to run spontaneously and transparently from the user's point of view. So, we want the reconfigure-TMs to be positioned in the tree as children of the user transactions, but we do not want the user programs to be aware of their invocations and returns.

This conflict introduces a modelling problem. We solve the problem by associating a *spy* automaton with each user transaction. The spy wakes up with the associated transaction and nondeterministically invokes reconfigure-TMs until the associated transaction requests to commit. In this way, we capture formally the notions of spontaneity and transparency while at the same time modelling the proper atomicity requirements.

Gifford's reconfiguration algorithm works as follows.[1] In addition to a value and a version number, each replica of x contains a configuration and a generation number. The value and version number are initialized as in the non-reconfiguration case, and all replicas of x initially hold the same configuration and generation number.

To perform a logical read of x, a TM reads DMs for x, keeping in its state the value v and version number t from the DM with the highest version number seen, the configuration c and generation number g from the copy with the highest generation number seen, and the set d of the names of the DMs read. If the TM reaches a state in which c has a read-quorum that is a subset of d, then the TM returns v.

To perform a logical write of x with new value v', a TM again reads DMs for x, keeping in its state the version number t from the DM with the highest version number seen, the configuration c and generation number g from the DM with the highest generation number seen, and the set d of the names of the DMs read. If the TM reaches a state in which c has a read-quorum that is a subset of d, then the TM computes the new version number $t' = t + 1$ and writes v' along with t' to some write-quorum of DMs in c.

To reconfigure x with new configuration c', a TM first reads DMs for x and computes v, t, c, g, and d, just as for a logical read. If the TM reaches a state in which c has a read-quorum that is a subset of d, then the TM does the following. It writes v and t to a write-quorum in c', and it writes c' and $g' = g + 1$ to a write-quorum in c.[2]

We generalize Gifford's reconfiguration algorithm in the same ways that we generalized the fixed quorum consensus algorithm in the previous chapter. A formal description of the

[1]In [Gi], Gifford describes the algorithm in terms of *votes*. However, we substitute the more general configuration definition.

[2]The description in [Gi] actually requires that the new configuration be written to both an old and a new write-quorum. However, we find that it is only necessary to write this information to an old write-quorum.

generalized algorithm follows. Because of the additional complication involved in reconfiguration and in order to avoid needless repetition of code, we separate the read, write, and reconfigure tasks of the TMs into modules called ***coordinators***. This is done most naturally by introducing another level of nesting, providing additional evidence of the power of nesting as a modelling tool.

The formalisms and proofs of this section follow the same pattern as those of the previous section.

4.1 Reconfigurable Replicated Serial System

Like the fixed configuration system, the replicated serial system defined in this section is an ordinary serial system in which certain logical data items are replicated. We impose a restriction on the transaction tree that all accesses to the replicas are the children of coordinator automata, which are, in turn, children of TMs. Together, the coordinators and TMs model the Quorum Consensus algorithm itself. We model the logical operations of the algorithm by providing three kinds of TMs: read-TMs, write-TMs, and reconfigure-TMs. The system type is formally defined as follows.

Fix I, a set of logical data items. We define system B to be a serial system of type $\langle \mathcal{T}, \text{parent}, \mathcal{O}, V \rangle$. For each element x of I, we define:

- $dm(x)$, a subset of $\mathcal{O}$,
- $acc(x)$, a subset of the accesses in $\mathcal{T}$,
- $co(x)$, a subset of the non-accesses in $\mathcal{T}$,
- $tm_r(x)$, $tm_w(x)$, and $tm_{rec}(x)$, disjoint subsets of the non-accesses in $\mathcal{T}$,
- $config(x)$, a legal configuration of x.

Let $tm(x) = tm_r(x) \cup tm_w(x) \cup tm_{rec}(x)$. We require that $acc(x)$ is exactly the set of all accesses to objects in $dm(x)$. Also, we require that T$\in acc(x)$ iff parent(T)$\in co(x)$, and that T$\in co(x)$ iff parent(T)$\in tm(x)$. That is, the accesses to DMs for x are exactly the children of the coordinators for x, which are, in turn, exactly the children of the TMs for x. Finally, for all pairs $x, y \in \mathcal{I}$, we require that $dm(x) \cap dm(y) = \emptyset$. As a notational convenience, we sometimes drop the "(x)" in $dm(x)$, $acc(x)$, etc. to denote the union of these sets over all $x \in \mathcal{I}$. For example, tm_{rec} is the union over all $x \in \mathcal{I}$ of $tm_{rec}(x)$.

The set of ***user transactions*** in system B consists of all non-access transactions that are neither in co nor tm. We refer to accesses in acc as ***replica accesses***, and to the remaining accesses in $\mathcal{T}$ as ***non-replica accesses***.

Figure 4.1 provides an example of a possible transaction tree for system B.

Each member of $dm(x)$ has an associated DM automaton for x. Each member of $co(x)$ has an associated read-coordinator, write-coordinator, or reconfigure-coordinator automaton for x. The members of $tm_r(x)$, $tm_w(x)$, and $tm_{rec}(x)$ have associated read-TM, write-TM, and reconfigure-TM automata for x, respectively.

Each user transaction T has an associated automaton that is the composition of a ***user automaton*** and a ***spy automaton*** T_{spy}. The user automaton may be any arbitrary automaton that satisfies the definition of a transaction, and does not have REQUEST-CREATE(T'), COMMIT for T', or ABORT(T') operations defined for any reconfigure-TM T'. The set ***spies*** refers to the collection of all spy automata in system B.

To avoid confusion, the reader should note that "user transaction" refers to the *name* in $\mathcal{T}$, whereas "user transaction automaton" refers to the automaton itself (the composition of the user automaton and the spy automaton).

To access x, a transaction invokes some read- or write-TM in $tm(x)$. This TM invokes one or more coordinators, each of which invokes read or write accesses to multiple DMs. The definitions constrain $\mathcal{T}$ so that all accesses to x must proceed in this fashion; no high-level transaction, for example, can directly invoke a coordinator for x or an access to a DM for x. DMs, coordinators, TMs, and spy automata are described in the next four subsections.

4.1.1 Data Managers

As before, the set of data managers for logical data item x models the set of physical replicas of x. Each DM is a read-write object that keeps a version-number, a value, a

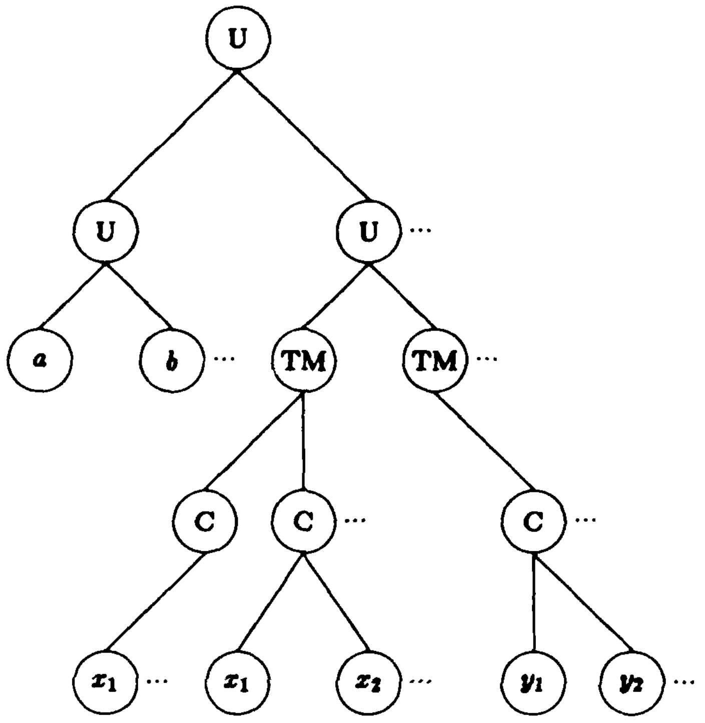


Figure 4.1: A possible transaction tree for system B. Transactions are labeled as follows: U = user transaction; TM = transaction manager; C = coordinator; a, b = non-replica accesses; x_1 = access to replica 1 of logical data item x, etc.

generation-number, and a configuration for x. The formal definition follows.

If x is a logical data item in I, a *DM* for x in system B is a read-write object over domain $D_x = N \times V_x \times N \times \text{legal}(dm(x))$ and with initial data $\langle 0, i_x, 0, config(x) \rangle$. We refer to each member of D_x as a ⟨version-number, value, generation-number, configuration⟩ quadruple.

Lemma 12 DMs are basic objects.

Proof: Immediate from Lemma 2. ∎

4.1.2 Coordinators

We now define the coordinators, which form the intermediate level of nesting between the replica accesses and the TMs. There are three types of coordinators: read, write, and reconfigure. Following the three coordinator definitions, we will define the TMs.

Read Coordinators: Let x be a logical data item in I. The purpose of a read-coordinator is to calculate the "current" version-number, value, generation-number, and configuration of x on the basis of the data returned by the read accesses it invokes.

Read-coordinator T has state components awake, data, requested, and read, where awake is a boolean variable, data is in the domain D_x, requested is a subset of $acc(x)$, and read is a subset of $dm(x)$. Initially, awake is false, data is $\langle 0, \perp, 0, config(x) \rangle$, and requested and read are both empty.

Input operations:	CREATE(T)
	COMMIT(T',v), where T' $\in$ children(T) and v $\in D_x$
	ABORT(T'), where T' $\in$ children(T)
Output operations:	REQUEST-CREATE(T'), where T' $\in$ children(T)
	REQUEST-COMMIT(T,v), where v $\in D_x$

```
• CREATE(T)
  Postcondition:  awake(s) = true

• REQUEST-CREATE(T'), where kind(T') = read
  Precondition:   awake(s') = true
                  T' ∉ requested(s')
  Postcondition:  requested(s) = requested(s') ∪ {T'}

• COMMIT(T',v)
  Postcondition:  read(s) = read(s') ∪ {O(T')}
                  if (v.version-number > data(s').version-number) then
                      data(s).version-number = v.version-number
                      data(s).value = v.value
                  if (v.generation-number > data(s').generation-number) then
                      data(s).generation-number = v.generation-number
                      data(s).configuration = v.configuration
```

- ABORT(T')

  Postcondition: (no change)

- REQUEST-COMMIT(T,v)

  Precondition: **awake(s') = true**
  $q \in$ **data(s')**.configuration.r
  $q \subseteq$ **read(s')**
  v = **data(s')**

  Postcondition: **awake(s) = false**

A read-coordinator collects data from DMs for x, and keeps track of the configuration from the DM with the highest generation number and the value from the DM with the highest version number seen so far. Whenever the read-coordinator reaches a state in which some read-quorum in the current configuration (i.e., some member of data(s').configuration.r) is a subset of the DMs it has seen (i.e., read(s')), then the read-coordinator may request to commit and return its data. The reader should compare the code above with the code for read-TMs in Chapter 3.

Write Coordinators: Let x be a logical data item in I. The purpose of a write-coordinator is to write a given value to a write-quorum of DMs for x in a given configuration of $dm(x)$.

A write-coordinator T has state components awake, requested, and written, where awake is a boolean variable, requested is a subset of $acc(x)$, and written is a subset of $dm(x)$. Initially, awake is false and the sets are empty. Every write-coordinator T for x has an associated value *value(T)* $\in V_x$, an associated version-number *version-number(T)* $\in N$, and an associated configuration *configuration(T)* $\in$ *legal*$(dm(x))$.

Input operations:	CREATE(T)
	COMMIT(T',v), where T' ∈ children(T)
	ABORT(T'), where T' ∈ children(T)
Output operations:	REQUEST-CREATE(T'), where T' ∈ children(T)
	REQUEST-COMMIT(T,v), where v = nil

- CREATE(T)

  Postcondition: awake(s) = true

- REQUEST-CREATE(T'), where kind(T') = write and data(T') = d

  Precondition: awake(s') = true
  d = ⟨version-number(T),value(T),⊥, ⊥⟩
  T' $\notin$ requested(s')

  Postcondition: requested(s) = requested(s') $\cup$ {T'}

- COMMIT(T',v)

  Postcondition: written(s) = written(s') $\cup$ {O(T')}

- ABORT(T')

  Postcondition: (no change)

- REQUEST-COMMIT(T,v)

  Precondition: awake(s') = true
  v = nil
  q $\in$ configuration(T).w
  q $\subseteq$ written(s')

  Postcondition: awake = false

When created, a write-coordinator begins invoking write accesses to DMs for x, overwriting the version-numbers and values at the DMs with its version-number and value, but leaving the generation-numbers and configurations at the DMs unchanged. After writing to a write-quorum of DMs according to its configuration, the write-coordinator may request to commit.

Reconfigure Coordinators: Let x be a logical data item in I. The purpose of a reconfigure-coordinator is to write a given new configuration for x along with a given generation number to a write-quorum of DMs in a given old configuration for x.

A reconfigure-coordinator T has state components awake, requested, and written, where awake is a boolean variables, requested is a subset of $acc(x)$, and written is a subset of $dm(x)$. Initially, awake is false and the sets are empty. Every reconfigure-coordinator T for x has associated configurations *new-configuration*(T), *old-configuration*(T) $\in$ *legal*($dm(x)$), and an associated generation-number *generation-number*(T) $\in N$.

Input operations:	CREATE(T)
	COMMIT(T',v), where T' ∈ children(T)
	ABORT(T'), where T' ∈ children(T)
Output operations:	REQUEST-CREATE(T'), where T' ∈ children(T)
	REQUEST-COMMIT(T,v), where v = nil

- CREATE(T)

  Postcondition: awake(s) = true

- REQUEST-CREATE(T'), where kind(T') = write and data(T') = d

  Precondition: awake(s') = true
  d = $\langle \perp,$ *perp*,generation-number(T), new-configuration(T) $\rangle$
  T' ∉ requested(s')

  Postcondition: requested(s) = requested(s') ∪ {T'}

- COMMIT(T',v)

  Postcondition: written(s) = written(s') ∪ {O(T')}

- ABORT(T')

  Postcondition: (no change)

- REQUEST-COMMIT(T,v)

  Precondition: awake(s') = true
  v = nil
  q ∈ old-configuration(T).w
  q ⊆ written(s')

  Postcondition: awake = false

When created, a reconfigure-coordinator begins invoking write accesses to the DMs for x, writing its generation-numbers and new-configurations to the DMs, but leaving the version numbers and values unchanged. When an old write-quorum of DMs has been written, according to its old-configuration, the reconfigure-coordinator may request to commit. This is an optimization over Gifford's algorithm. Gifford requires that the new configuration be written a new write-quorum, as well as to an old write-quorum.

Lemma 13 Coordinators are transactions.

Proof: The proof is identical to that of Lemma 4. ■

4.1.3 Transaction Managers

We now define the three kinds of TMs: read, write, and reconfigure. Read- and write-TMs are invoked by user automata in order to perform logical reads and writes to logical data items. Reconfigure-TMs are invoked by spy automata, which are defined following the TM definitions.

Read TMs: Let x be a logical data item in I. The purpose of a read-TM is to perform a logical read access to x on behalf of a user transaction.

Read-TM T has state components awake, data, requested, and read, where awake and read are boolean variables, data is in the domain D_x, and requested is a subset of $co(x)$. Initially, the booleans are false, data is undefined, and requested is empty.

Input operations:	CREATE(T)
	COMMIT(T',v), where T' $\in$ children(T) and v $\in D_x$
	ABORT(T'), where T' $\in$ children(T)
Output operations:	REQUEST-CREATE(T'), where T' $\in$ children(T)
	REQUEST-COMMIT(T,v), where v $\in D_x$

- CREATE(T)
  Postcondition: awake(s) = true

- REQUEST-CREATE(T'), where T' is a read-coordinator
  Precondition: awake(s') = true
  read(s') = false
  T' $\notin$ requested(s')
  Postcondition: requested(s) = requested(s') $\cup$ {T'}

- COMMIT(T',v)
  Postcondition: if read(s') = false then
      data(s) = v
      read(s) = true

- ABORT(T')
  Postcondition: (no change)

- REQUEST-COMMIT(T,v)
  Precondition: awake(s') = true
  read(s') = true
  v = data(s').value
  Postcondition: awake(s) = false

A read-TM invokes any number of read-coordinators. After one or more these coordinators commits, the read-TM may commit, returning the value component of the data returned by the first committing read-coordinator.

Write TMs: Let x be a logical data item in I. The purpose of a write-TM for x is to perform a logical write access to x on behalf of a user transaction.

Write-TM T has state components awake, data, requested, read, and written, where awake, read, and written are boolean variables, data is in the domain D_x, and requested is a subset of $co(x)$. Initially, the booleans are false, data is undefined, and requested is empty. Every write-TM T has an associated value *value*(T).

Input operations:	CREATE(T)
	COMMIT(T',v), where T' $\in$ children(T) and v $\in$ D_x
	ABORT(T'), where T' $\in$ children(T)
Output operations:	REQUEST-CREATE(T'), where T' $\in$ children(T)
	REQUEST-COMMIT(T,v), where v = nil

- CREATE(T)
  Postcondition: awake(s) = true

- REQUEST-CREATE(T'), where T' is a read-coordinator
  Precondition: awake(s') = true
  T' $\notin$ requested(s')
  Postcondition: requested(s) = requested(s') $\cup$ {T'}

- COMMIT(T',v), where T' is a read-coordinator
  Postcondition: if read(s') = false then
    data(s) = v
    read(s) = true

- REQUEST-CREATE(T'), where T' is a write-coordinator
  Precondition: awake(s') = true
  read(s') = true
  value(T') = value(T)
  version-number(T') = data(s').version-number+1
  configuration(T') = data(s').configuration
  T' $\notin$ requested(s')
  Postcondition: requested(s) = requested(s') $\cup$ {T'}

- COMMIT(T',v), where T' is a write-coordinator
  Postcondition: written(s') = true

- ABORT(T')
  Postcondition: (no change)

- REQUEST-COMMIT(T,v)
  Precondition: awake(s') = true
  written(s') = true
  v = nil
  Postcondition: awake(s) = false

A write-TM invokes some number of read-coordinators; when the first read-coordinator commits, the write-TM remembers the data returned. The write-TM then has the option of invoking any number of write coordinators, using the configuration and version-number (incremented by one) it remembered from the first committing read-coordinator, along with its particular data value. In order for the write-TM to commit, at least one of the write-coordinators must have committed.

Reconfigure TMs: Let x be a logical data item in I. The purpose of a reconfigure-TM is to change the "current" configuration of x to a given target configuration and to propagate the current value and version number as necessary.

Reconfigure-TM T has state components awake, data, requested, read, and written, where awake, read and written are boolean variables, data is in the domain D_x, and requested is a subset of $co(x)$. Initially, the booleans are false, data is undefined, and requested is empty. Every reconfigure-TM T has an associated configuration ***target-configuration***(T) $\in$ legal($dm(x)$).

Input operations:	CREATE(T)
	COMMIT(T',v), where T' $\in$ children(T) and v $\in D_x$
	ABORT(T'), where T' $\in$ children(T)
Output operations:	REQUEST-CREATE(T'), where T' $\in$ children(T)
	REQUEST-COMMIT(T,v), where v = nil

- CREATE(T)

```
Postcondition:  awake(s') = true
```

- REQUEST-CREATE(T'), where T' is a read-coordinator

```
Precondition:   awake(s') = true
                T' ∉ requested(s')
Postcondition:  requested(s) = requested(s') ∪ {T'}
```

- COMMIT(T',v), where T' is a read-coordinator

```
Postcondition:  if read(s') = false then
                    data(s) = v
                    read(s) = true
```

- REQUEST-CREATE(T'), where T' is a write-coordinator

```
Precondition:   awake(s') = true
                read(s') = true
                value(T') = data(s').value
                version-number(T') = data(s').version-number
                configuration(T') = target-configuration(T)
                T' ∉ requested(s')
Postcondition:  requested(s) = requested(s') ∪ {T'}
```

- COMMIT(T',v), where T' is a write-coordinator

```
Postcondition:  written(s') = true
```

- REQUEST-CREATE(T'), where T' is a reconfigure-coordinator

Precondition:	awake(s') = true
	read(s') = true
	old-configuration(T') = data(s').configuration
	new-configuration(T') = target-configuration(T)
	generation-number(T') = data(s').generation-number+1
	T' $\notin$ requested(s')
Postcondition:	requested(s) = requested(s') $\cup$ {T'}

- COMMIT(T',v), where T' is a reconfigure-coordinator

  Postcondition: (no change)

- ABORT(T')

  Postcondition: (no change)

- REQUEST-COMMIT(T,v)

Precondition:	awake(s') = true
	written(s') = true
	v = nil
Postcondition:	awake(s) = false

A reconfigure-TM invokes some number of read-coordinators; when the first read-coordinator commits, the reconfigure-TM remembers the data returned. Then, the reconfigure-TM may invoke any number of write coordinators with its target-configuration, and with the value and version number from the data returned by the first committing read-coordinator. Also, the TM may invoke any number of reconfigure-coordinators, where the old configuration and old generation number are those from the first committing read-coordinator and the new configuration is the TM's target-configuration. In order to commit, at least one write-coordinator must have committed.[3]

Lemma 14 TMs are transactions.

Proof: The proof is identical to that of Lemma 4. ■

Recall that reconfigure-TMs are invoked only by spy automata, which are composed with user automata to form user transaction automata. (See Lemma 15.) Spy automata are defined as follows.

[3]There is no need to wait for a reconfigure-coordinator to commit: Should all the reconfigure-coordinators abort, the reconfigure-TM would have merely propagated the current value.

4.1.4 Spy Automata

The spy automaton T_{spy} associated with user transaction T has state components awake and create-requested, where awake is a boolean and create-requested is a subset of tm_{rec}. Initially, awake is false and create-requested is empty.

Input operations:	CREATE(T)
	COMMIT(T',v), where T' $\in$ children(T)
	ABORT(T'), where T' $\in$ children(T)
	REQUEST-COMMIT(T,v)
Output operations:	REQUEST-CREATE(T'), where T' $\in$ children(T)

- CREATE(T)

  Postcondition: awake(s) = true

- REQUEST-CREATE(T'), where T' is a reconfigure-TM

  Precondition: awake(s') = true
  T' $\notin$ create-requested(s')
  Postcondition: create-requested(s) = create-requested(s') $\cup$ {T'}

- COMMIT(T',v)

  Postcondition: (no change)

- ABORT(T')

  Postcondition: (no change)

- REQUEST-COMMIT(T,v)

  Postcondition: awake(s) = false

Note that a spy is *not* a transaction automaton, but rather one piece of a transaction. It wakes up when its associated transaction T is created and goes to sleep when T requests to commit. That is, the spy automaton does not request to commit; instead, it receives REQUEST-COMMIT(T,v) as an *input* operation.

While a spy automaton is awake, it may invoke any number of reconfigure-TMs. In this way, the model formalizes the spontaneous invocation of reconfigure-TMs. The user automaton associated with a given spy has no control over the configurations of the logical data items. In fact, the user automaton cannot directly observe changes in configurations because it has no input operations that could reveal this information.

It is interesting that in our definition of the spy automaton, the choice of when to change configuration and which new configuration to use is completely general (i.e., nondeterministic). However, one might wish to add a heuristic for making judicious choices about when and how to intervene. This may involve adding new input operations and state components to the spy that would allow it to record the CREATE, COMMIT, and ABORT patterns of accesses to the logical data items. The nondeterminism of the spy automaton allows such heuristics to be added without compromising the validity of our results.

Lemma 15 Let user automaton T be a transaction that does not have any REQUEST-CREATE(T') operations defined where T' is a reconfigure-TM. Then the composition, also named T, of user automaton T with spy automaton T_{spy} is also a transaction.

Proof: It suffices to show that the composition preserves well-formedness. Let $\alpha = \alpha'\pi$ be a schedule of the composition where π is an output operation of the composition, and assume that α' is well-formed. We need to show that: (1) CREATE(T) occurs in α', (2) no REQUEST-COMMIT operation occurs in α', and (3) if π is a REQUEST-CREATE(T') operation, then no REQUEST-CREATE(T') occurs in α'.

Since user automaton T is a transaction, we know that it cannot issue an output operation unless CREATE(T) occurs in α'. By definition, T_{spy} can issue no output operation unless its awake flag is true, and only a CREATE(T) operation can set awake to true. Therefore, neither user automaton T nor T_{spy} can issue an output operation unless CREATE(T) occurs in α'. So, part (1) holds for the composition.

Similarly, for part (2), we know that if π is an output operation of user automaton T then no REQUEST-COMMIT for T occurs in α' because the user automaton is a transaction and T_{spy} does not issue REQUEST-COMMIT for T operations. T_{spy} can issue no output operation if the awake flag is false, and only a REQUEST-COMMIT for T operation can cause the awake flag to become false. Since α' is well-formed, it can contain at most one CREATE(T) operation. So, once awake becomes false, it is false forever. Therefore, neither user automaton T nor T_{spy} can issue an output operation if REQUEST-COMMIT for T occurs in α'. Thus, part (2) holds for the composition.

Finally, we note that user automaton T does not invoke reconfigure-TMs and T_{spy} invokes only reconfigure-TMs. So, it is sufficient to show that part (3) holds for user automaton T and T_{spy} independently. We know that part (3) holds for the user automaton, because it is a transaction. Whenever it issues a REQUEST-CREATE(T'), T_{spy} puts T' into its create-requested list, and nothing is ever removed from the create-requested list. Since a precondition for REQUEST-CREATE(T') is that T' is not in the create-requested list, part (3) must hold for T_{spy}. Therefore, part (3) holds for the composition. ■

4.1.5 Properties

In this subsection, we prove several interesting properties of system B. Most of the subsection is devoted to the proof of Lemma 20, which is central to our correctness argument.

Lemma 16 Schedules of system B are well-formed.

Proof: By Lemmas 12, 13, and 14, DMs are basic objects, and coordinators and TMs are transactions. Furthermore, by Lemma 15, all the remaining members of $\mathcal{T}$ are transactions. Therefore, system B is a serial system. By [LM], all schedules of serial systems are well-formed. ■

We now present definitions to describe the logical accesses to the logical data items in system B. These definitions are analogous to those for the fixed configuration system.

Access sequence: Let β be a sequence of operations of system B, and let x be a logical data item in $\mathcal{I}$. Then the *access sequence* of x in β, denoted *access*(x, β), is defined to be the subsequence of β containing the CREATE and REQUEST-COMMIT operations for the members of *tm*(x).

Logical state: Let β be a sequence of operations of system B, and let x be a logical data item in $\mathcal{I}$. The *logical state of x after β*, denoted *logical-state*(x, β), is defined to be either *value*(T) if REQUEST-COMMIT(T,v) is the last REQUEST-COMMIT operation for a write-TM in access(x, β), or i_x if no REQUEST-COMMIT operation for a write-TM occurs in access(x, β).

Current version number: Let β be a sequence of operations of system B, and let x be a logical data item in I. Let *last*(x, β) denote the subset of *acc*(x) such that for each member T of last(x, β), REQUEST-COMMIT for T is the last REQUEST-COMMIT operation for a write access to O(T) in β with data(T).version-number$\neq\perp$[4]. The *current version number of x after β*, denoted *current-vn*(x, β), is defined as follows. If last(x, β) is non-empty, then current-vn(x, β) is the maximum over all T$\in$last(x, β) of data(T).version-number. Otherwise, current-vn(x, β) = 0.

Current generation number: Let β be a sequence of operations of system B, and let x be a logical data item in I. Let *last*(x, β) denote the subset of *acc*(x) such that for each member T of last(x, β), REQUEST-COMMIT for T is the last REQUEST-COMMIT operation for a write access to O(T) in β with data(T).generation-number$\neq\perp$[5]. The *current generation number of x after β*, denoted *current-gn*(x, β), is defined as follows. If last(x, β) is non-empty, then current-gn(x, β) is the maximum over all T$\in$last(x, β) of data(T).generation-number. Otherwise, current-gn(x, β) = 0.

Lemma 17 If β is a schedule of B and x is a logical data item in I, then access(x, β) begins with a CREATE operation for some TM in *tm*(x) and continues alternately with REQUEST-COMMIT and CREATE operations for TMs in *tm*(x) such that each REQUEST-COMMIT for T is preceded immediately by a CREATE(T) operation.

Proof: By definition, access(x, β) contains only CREATE and REQUEST-COMMIT operations for TMs in *tm*(x). By Lemma 16, β is a well-formed schedule, so each REQUEST-COMMIT for T must be preceded by a CREATE(T) operation. Finally, since β is a serial schedule, all operations for a given transaction must be contiguous. ■

[4]This last condition allows us to consider only those write accesses which change the version number at a DM

[5]Here, we are only interested in those write accesses which change the generation number at a DM

Lemma 18 Let x be a logical data item, and let β be a schedule of B. Then the following property holds after β: The highest version number among the states of all DMs in $dm(x)$ is current-vn(x,β).

Proof: Since DMs are read-write objects, the only operation that can change the version-number in the state of a DM O for x is a REQUEST-COMMIT for T operation, where O(T) $= O$ and T is a write access with data(T).version-number$\neq\perp$. More specifically, the version-number in the state of a DM O after β is data(T).version-number, where REQUEST-COMMIT for T is the last such REQUEST-COMMIT in β. In the definition of current-vn(x,β), the set last(x,β) contains the last write access with version-number$\neq\perp$ for each DM in $dm(x)$ that has a REQUEST-COMMIT for such a write access in β. Therefore, the maximum over all T$\in$last(x,β) of data(T).version-number is the highest version number among the states of all DMs in $dm(x)$ after β. This maximum is exactly the definition of current-vn(x,β). (If no such REQUEST-COMMITs occur in β, then all DMs have their initial version-number, which is 0 by definition.) ■

Lemma 19 Let x be a logical data item, and let β be a schedule of B. Then the following property holds after β: The highest generation number among the states of all DMs in $dm(x)$ is current-gn(x,β).

Proof: Analogous to that of Lemma 18. ■

The main lemma is again proved by induction on the length of β. We take advantage of the nesting structure in the proof by proving simple assertions about the sub-transactions of the TMs, and then using these simple assertions to prove the main assertions about the TMs. As before, the first condition is only used for carrying through the inductive argument. The important part of the lemma is the second condition, which tells us that read-TMs return the proper value.

Lemma 20 Let x be a logical data item in $\mathcal{I}$. Let β be a schedule of B such that access(x,β) is of even length.

1. The following properties hold after β:

(a) For all DMs $O \in dm(x)$, if d is the data component of O, and d.generation-number < current-gn(x,β), then there exists some write-quorum $q \in$ d.configuration.w such that for all DMs $O' \in q$, if d' is the data component of O' then d'.generation-number > d.generation-number.

(b) For all pairs of DMs $O_1, O_2 \in dm(x)$, if d_1 and d_2 are the data components of O_1 and O_2, then d_1.generation-number = d_2.generation-number implies that d_1.configuration = d_2.configuration. Let *logical-config*(x,β) denote the unique configuration held by all DMs with generation-number = current-gn(x,β).

(c) There exists a write-quorum $q \in$logical-config(x,β).w such that for all DMs $O \in q$, if d is the data component of O, then d.version-number = current-vn(x,β).

(d) For all DMs $O \in dm(x)$, if d is the data component of O, then d.version-number = current-vn(x,β) implies that d.value = logical-state(x,β).

2. If β ends in REQUEST-COMMIT(T,v) with T$\in tm_r(x)$, then v = logical-state(x,β).

Proof: By induction on the length of β.

Base case: Let β be the empty schedule. By definition, current-gn$(x,\beta) = 0$, current-vn$(x,\beta) = 0$, and logical-state$(x,\beta) = i_x$. Initially, all DMs in $dm(x)$ have generation-number = 0, configuration = *config*(x), version-number = 0 and value = i_x by the definition of a DM. Therefore, logical-config(x,β) = *config*(x). Furthermore, the states after β of all the DMs in every $q \in$ *config*(x).w have generation-number = current-gn(x,β), configuration = *config*(x), version-number = current-vn(x,β), and value = logical-state(x,β). Thus, part 1 holds. Since β is empty, it does not end in a REQUEST-COMMIT operation of a read-TM for x. So, part 2 holds vacuously.

Induction: Let $\beta = \beta'\tau$, where access(x,τ) begins with the last CREATE operation in access(x,β). Assume that the Lemma holds for β'. By Lemma 17 and the fact that access(x,β) is of even length, access(x,τ) = (CREATE(T_f), REQUEST-COMMIT(T_f,v_f)) for some $T_f \in tm(x)$ and $v_f \in V_x$. We note the following facts about T_f:

Fact 1: All accesses in τ to DMs in $dm(x)$ are descendants of T_f.

Proof: Since β is a serial schedule, T_f is the only TM in $tm(x)$ whose descendants have operations in r. Furthermore, the system type of B is constrained so that all accesses to DMs in $dm(x)$ are descendants of TMs in $tm(x)$.

Since T_f requests to commit in r, we know by definition of T_f that at least one read-coordinator, a child of T_f, must commit to T_f in r. Let T' be the first read-coordinator that commits to T_f in r, and let r' be the portion of r up to and including the COMMIT for T'.

Fact 2: If s is the state of T' just after a read access commits to T' in r', then

1. data(s).generation-number and data(s).configuration contain the highest generation-number and associated configuration among DMs in read(s), and
2. data(s).version-number and data(s).value contain the highest version-number and associated value among DMs in read(s).

Proof: This fact holds because T' retains the maximum generation-number and version-number (seen so far) and their respective configuration and value upon each commit of a read access. Since T' is the first child that commits to T_f and since T' invokes only read accesses, the data components of all DMs observed by T' must be the same during r' as after β'.

Fact 3: The data component of the state of T_f forever after $\beta' r'$ is $\langle$current-vn(x, β'), logical-state(x, β'), current-gn(x, β'), logical-config$(x, \beta')\rangle$.

Proof: Let s' be the state of T' when T' issues its REQUEST-COMMIT operation. Together, part 1 of Fact 2 and part (1a) of the induction hypothesis imply that read(s') cannot contain a read-quorum according to data(s').configuration unless data(s').generation-number = current-gn(x, β'). By definition, T' cannot commit unless read(s') contains a read-quorum in data(s').configuration.r. Therefore, by part (1b) of the induction hypothesis, we know that read(s') contains some read-quorum r in data(s').configuration = logical-config(x, β').r. By part (1c) of the induction hypothesis, we know that there exists some write-quorum $w \in$ logical-config(x, β').w such that the states after β' of all DMs in w have version-number = current-vn(x, β'). Since

logical-config(x, β') is a legal configuration, r and w must have a non-empty intersection. So, by part 2 of Fact 2, data(s').version-number = current-vn(x, β'). Therefore, by part (1d) of the induction hypothesis, data(s').value = logical-state(x, β').
When T' commits, the data component of the state of T_f becomes data(s'). By definition, once T' commits, the data component of the state of T_f never changes. Therefore, Fact 3 is proved.

From Fact 1, we know that all accesses to DMs for x in τ are children of T_f. Therefore, in order to prove that the induction hypothesis holds for β, we merely need to demonstrate that T_f preserves the properties stated. There are three possibilities for T_f:

- If T_f is a read-TM, then T_f invokes only read-coordinators, which invoke only read accesses. So, current-vn(x, β) = current-vn(x, β') and current-gn(x, β) = current-gn(x, β'). Furthermore, the data components of the states of the DMs are the same after β as after β'. Therefore, part 1 of the Lemma holds for β. By definition, T_f cannot request to commit until at least one of its read-coordinators commits. Since T' is the first committing read-coordinator, the REQUEST-COMMIT for T_f must occur at some point after $\beta'\tau'$. When T_f commits, it returns the value in the data component of its state. By Fact 3, this value is logical-state(x, β'). Since T_f is a read-TM, logical-state(x, β) = logical-state(x, β') by definition. Thus, part 2 of the Lemma holds for β.

- If T_f is a write-TM, then we note the following facts:

  Fact 4: All write-coordinators T for x invoked in τ have version-number(T) = current-vn(x, β')+1, value(T) = value(T_f), and configuration(T_w) = logical-config(x, β').

  Proof: Let s be the state of T_f when it issues REQUEST-CREATE(T). Then by definition of a write-TM, version-number(T) = data(s).version-number+1, value(T) = value(T_f), and configuration(T) = data(s).configuration. By definition, T_f cannot invoke a write-coordinator until at least one of its read-coordinators commits. So, all REQUEST-CREATEs for write-coordinators in τ

occur after $\beta'\tau'$. Therefore, by Fact 3, data(s).version-number = current-vn(x, β') and data(s).configuration = logical-config(x, β'). Thus, Fact 4 holds.

Fact 5: If T is a write access for x invoked in τ, then data(T) = $\langle$current-vn(x, β), logical-state(x, β),$\perp, \perp\rangle$, and current-vn(x, β) > current-vn(x, β').

Proof: The type of system B is constrained so that T is invoked by some write-coordinator for x. Therefore, by Fact 4 and the definition of a write-coordinator, data(T) = $\langle$current-vn(x, β')+1, value(T_f),$\perp, \perp\rangle$. Therefore, since current-vn(x, β')+1 is the highest (only) version-number for x written in τ, it follows from Lemma 18 and the definition of current-vn that current-vn(x, β) = current-vn(x, β')+1. Since T_f is a write-TM, logical-state(x, β) = value(T_f). Thus, Fact 5 is proved.

By Fact 5, the generation-numbers and configurations in the states of DMs for x are not changed during τ, and current-gn(x, β) = current-gn(x, β'). Therefore, parts (1a) and (1b) of the Lemma hold after β. (Note that logical-config(x, β') = logical-config(x, β).)

By definition, T_f cannot request to commit until at least one of its write-coordinators commits. Let T_w be the first write-coordinator that commits to T_f, and let τ'' be the portion of τ up to and including the COMMIT of T_w. By definition, T_w cannot request to commit until it has received COMMITs for write accesses to a write quorum of DMs in configuration(T_w). By Fact 4, configuration(T_w) = logical-config(x, β'), which equals logical-config(x, β). Therefore, by Fact 5, parts (1c) and (1d) of the Lemma hold after $\beta'\tau''$.

We now show that part 1 of the Lemma still holds after $\beta'\tau$. By Fact 5, any write-coordinators that may execute in τ after τ'' merely propagate the new value and version number. Any read-coordinators that may execute in τ after τ'' cannot change the values at the DMs, since they do not invoke write accesses. Therefore, part 1 of the Lemma holds after $\beta'\tau = \beta$.

Since T_f is not a read-TM, part 2 holds vacuously.

- If T_f is a reconfigure-TM, then we note the following facts:

Fact 6: All write-coordinators T_w for x invoked in τ have have version-number(T_w) = current-vn(x, β'), value(T_w) = logical-state(x, β'), and configuration(T_w) = logical-config(x, β'). Furthermore, all reconfigure-coordinators T_{rec} for x invoked in τ have generation-number(T_{rec}) = current-gn(x, β')+1, old-configuration(T_{rec}) = logical-config(x, β'), and new-configuration(T_{rec}) = target-configuration(T_f).

Proof: Analogous to that of Fact 4.

Fact 7: If T is a write access invoked by a write-coordinator for x in τ, then data(T) = $\langle$current-vn(x, β), logical-state(x, β), $\perp, \perp\rangle$.

Proof: By Fact 6 and the definition of a write-coordinator, data(T) = $\langle$current-vn(x, β'), logical-state(x, β'), $\perp, \perp\rangle$. Since these are the only write accesses in τ that modify the version-number components in the states of DMs for x, we know by Lemma 18 and the definition of current-vn that current-vn(x, β) = current-vn(x, β'). Since T_f is a reconfigure-TM, logical-state(x, β) = logical-state(x, β') by definition.

Fact 8: If T is a write access invoked by a reconfigure-coordinator for x in τ, then data(T) = $\langle\perp, \perp$,current-gn(x, β),logical-config(x, β) $\rangle$, and current-gn(x, β) > current-gn(x, β').

Proof: By Fact 6 and the definition of a reconfigure-coordinator, data(T) = $\langle\perp, \perp$, current-gn(x, β')+1,target-configuration(T_f)$\rangle$. Therefore, since current-gn(x, β')+1 is the highest (only) generation-number for x written in τ, it follows from Lemma 19 and the definition of current-gn that current-gn(x, β) = current-gn(x, β')+1. Also, target-configuration(T_f) is the configuration associated with current-gn(x, β), which is logical-config(x, β) by definition.

By definition, T_f cannot request to commit until at least one of its write-coordinators commits. Let T_w be the first write-coordinator that commits to T_f, and let τ'' be the portion of τ up to and including the COMMIT of T_w. We claim that part 1 of the induction hypothesis holds after $\beta'\tau''$. There are two cases:

1. If τ'' does not contain a COMMIT of a reconfigure-coordinator, then by Fact 7, any write accesses invoked in τ'' simply propagate the current value and version

number, so part 1 still holds.

2. If τ'' does contain one or more COMMITs of reconfigure-coordinators, then each reconfigure-coordinator T_{rec} cannot commit until it has received COMMIT operations for write accesses to a write-quorum w of DMs in old-configuration(T_{rec}). We now show that part (1a) of the lemma holds after $\beta'\tau''$. There are two classes of DMs to consider: (1) All DMs that have generation-number = current-gn(x, β') after $\beta'\tau''$ must have configuration = logical-config(x, β') by Fact 8 and part (1a) of the induction hypothesis. By Fact 6, old-configuration(T_w) = logical-config(x, β'). Therefore, $w \in$ logical-config(x, β').w, so part (1a) holds. (2) For all DMs that have generation-number < current-gn(x, β'), we know by Fact 8 and Lemma 19 that no DM's generation-number could have been decreased in τ''. So, by part (1a) of the induction hypothesis, part (1a) still holds.

   By Fact 8 and Lemma 19, we know that all write accesses for x in τ'' with generation-number$\neq\perp$ have a generation-number greater than (and therefore different from) any generation-number for x in β'. Furthermore, since all such write accesses have the same generation-number and configuration, we know by part (1b) of the induction hypothesis that part (1b) still holds. By definition, T_w cannot request to commit until it has received COMMIT operations for write accesses to a write quorum of DMs in configuration(T_w). Therefore, by Fact 7, parts (1c) and (1d) hold after $\beta'\tau''$.

Thus, claim is true in both cases, so part 1 of the lemma holds after $\beta'\tau''$.

By Fact 7, any write-coordinators that may execute in τ after τ'' merely propagate the new value and version number, so they preserve part 1 of the induction hypothesis. Similarly, by Fact 8, any reconfigure-coordinators that may execute in τ after τ'' merely propagate the new configuration and generation-number. And certainly any read-coordinators that may execute in τ after τ'' cannot change the data components of the DMs. Therefore, part 1 of the induction hypothesis holds when T_f commits.

Since T_f is not a read-TM, part 2 holds vacuously.

For all three possibilities of T_f, the lemma holds after β. ■

4.2 Non-replicated Serial System

We define non-replicated serial system A of type $(\mathcal{T}_A, \text{parent}_A, \mathcal{O}_A, \mathcal{V}_A)$ in terms of replicated serial system B of type $\langle \mathcal{T}_B, \text{parent}_B, \mathcal{O}_B, \mathcal{V}_B \rangle$. The transactions that were read-TMs and write-TMs for objects in $\mathcal{I}$ in system B become accesses in system A, and the collection of DMs for each object in $\mathcal{I}$ in system B are replaced by a single read-write object in system A. The reconfigure-TMs, coordinators and accesses from system B are not present in system A. More formally, the system type is:

- $\mathcal{T}_A = \mathcal{T}_B - \left(\bigcup_{x \in \mathcal{I}} acc(x)\right) - \left(\bigcup_{x \in \mathcal{I}} co(x)\right) - \left(\bigcup_{x \in \mathcal{I}} tm_{rec}(x)\right)$
- $\text{parent}_A = \text{parent}_B$ restricted to $\mathcal{T}_A$
- $\mathcal{O}_A = \mathcal{O}_B - \left(\bigcup_{x \in \mathcal{I}} dm(x)\right) \cup \{tm_r(x) \cup tm_w(x) \cup tm_{rec}(x) | x \in \mathcal{I}\}$
- $\mathrm{V}_A = \mathrm{V}_B$

Informally, to construct the type of system A from that of system B, we first remove from $\mathcal{T}$ all the coordinators, all the reconfigure-TMs, and all the accesses to the DMs for objects in $\mathcal{I}$. As a result, all the TMs for objects in $\mathcal{I}$ become leaves in $\mathcal{T}$ and are therefore accesses. Next, we remove from $\mathcal{O}$ all the DMs for objects in $\mathcal{I}$. (Effectively, this has already been done by removing the corresponding accesses.) Finally, we partition all the accesses that were formerly TMs according to their logical data item. Each class of this partition is a new object in $\mathcal{O}$. Thus, each logical data item is implemented by a single object.

Figure 4.2 illustrates the transaction tree for system A that corresponds to the transaction tree for system B given in Figure 4.1.

The following lemma tells us that that the function $\mathcal{F}_{AB}$ is well-defined. This allows us to relate transactions in system B to those in system A.

Lemma 21 System B is an extension of system A.

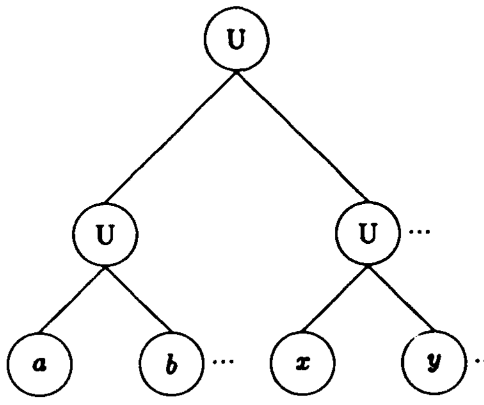


Figure 4.2: The transaction tree for system A that corresponds to the transaction tree for B shown in Figure 4.1. Transactions are labeled as follows:
U = user transaction; a, b, x, y = accesses.

Proof: Since $\mathcal{T}_A = \mathcal{T}_B - (\bigcup_{x\in I} acc(x)) - (\bigcup_{x\in I} co(x)) - (\bigcup_{x\in I} tm_{rec}(x))$, we know that $\mathcal{T}_A \subseteq \mathcal{T}_B$. Furthermore, $\mathcal{T}_A$ and $\mathcal{T}_B$ must have the same root, unless the root of $\mathcal{T}_B$ is in $acc(x)$, $co(x)$, or $tm_{rec}(x)$ for some $x \in I$. However, every member of $acc(x)$ is a child of some member of $co(x)$, which in turn is a child of some member of $tm(x)$. In addition, every member of $tm_{rec}(x)$ is a child of some user transaction. So none of the transactions in $acc(x)$, $co(x)$, or $tm_{rec}(x)$ for any $x \in I$ could be the root of $\mathcal{T}_B$. ∎

We define *user transactions* in system A to be all non-access transactions in $\mathcal{T}_A$. Just as for the fixed configuration systems, we note that T is a user transaction in system B iff $\mathcal{F}_{BA}$(T) is a user transaction in system A. Transactions and objects in system A are modelled in the same way as in system B, except that for all $x \in I$,

1. the object corresponding to $tm(x)$ has an associated read-write object O over domain V_x (We refer to this particular read-write object as O(x).),

2. for each transaction T$\in tm(x)$ where T is a read-TM or write-TM, $\mathcal{F}_{BA}$(T) is an access to O(x) such that

(a) if T is a read-TM, then $\mathcal{F}_{BA}(\mathrm{T})$ is a read-access, and

(b) if T is a write-TM, $\mathcal{F}_{BA}(\mathrm{T})$ is a write-access with data($\mathcal{F}_{BA}$(T)) = value(T).

Furthermore, if T is a user transaction, then it is modelled by the same user automaton as in system B, but *without* an associated spy automaton.

4.3 Correctness

In this section, we show that user transactions cannot distinguish between replicated serial system B and non-replicated serial system A. The proof is analogous to that of Theorem 10, but this time making use of Lemma 20. We are only interested in the correspondence between the schedules of the user automata in systems A and B. So, in Condition 2 of the theorem, we only require that the user automata, rather than the user transactions, have the same schedules.

Theorem 22 Let β be a schedule of replicated serial system B. There exists a schedule α of non-replicated serial system A such that the following two conditions hold.

1. For all objects O in system B that are not in $dm(x)$ for any x, $\alpha|\mathrm{O} = \beta|\mathrm{O}$.
2. For all user transactions T in system B, $\alpha|\mathcal{F}_{BA}(\mathrm{T}) = \beta|\mathrm{T}_{user}$, where T_{user} is the user automaton associated with T.

Proof: We construct α by removing from β all the REQUEST-CREATE(T), CREATE(T), REQUEST-COMMIT(T,v), COMMIT(T,v), and ABORT(T) operations for all transactions T in $acc(x)$, $co(x)$, and $tm_{rec}(x)$, for all $x \in \mathcal{I}$. Clearly, the two conditions hold. What needs to be proved is that α is a schedule of A. We proceed by induction on the length of β.

Base case: Suppose β is empty. Then α is also empty and is therefore a schedule of A.

Induction: Let $\beta = \beta'\pi_\beta$, where the claim holds for β'. Let $\alpha = \alpha'\pi_\alpha$, where α' is the schedule of A corresponding to β'. There are five cases for π_β.

read-write object, v' is the value in the state of $O(x)$ after α'. We observe that, by the construction, $\alpha'|O(x) = \text{access}(x, \beta')$. So, by definition of system A, the last write access in α' to $O(x)$ has the same value as the last write-TM in β'. Hence, the value in the state of $O(x)$ after α' is logical-state(x, β'). Therefore, $v = v'$. (If there is no write-TM in access(x, β'), then there are no write accesses to $O(x)$ in α'. In this case, the value in the state of $O(x)$ after α' is i_x, which is logical-state(x, β').)

5. **Output Operations of the Scheduler** (except those already covered by Case 1): If π_β is a CREATE(T), a COMMIT for T, or an ABORT(T), where T is a user transaction, T is a non-replica access, or T is in $tm_r(x)$ or $tm_w(x)$ for some $x \in I$, then by the construction $\pi_\alpha = \pi_\beta$.

   If π_β is a CREATE(T) or ABORT(T), then the preconditions for π_α are (1) there must be a REQUEST-CREATE(T) in α' but no CREATE(T) or ABORT(T) in α', and (2) all siblings of $\mathcal{T}_{BA}$(T) with creates in α' must have returned (committed or aborted) in α'. Since β is a well-formed schedule, REQUEST-CREATE(T) is in β', and, by the construction, is in α' as well. Similarly, since no CREATE(T) or ABORT(T) can occur in β', none can occur in α' either. Therefore, precondition (1) is satisfied. By the construction, all commits and aborts in α' of siblings of $\mathcal{T}_{BA}$(T) must also appear in β'. So, since β is well-formed, precondition (2) must also be satisfied.

   If π_β is a COMMIT(T,v), then the preconditions for π_α are (1) a REQUEST-COMMIT(T,v) must occur in α', (2) $\mathcal{T}_{BA}$(T) cannot have a COMMIT or ABORT in α', and (3) any children invoked by $\mathcal{T}_{BA}$(T) must have returned in α'. Using the same argument as above, by the construction and the fact that β is well-formed, preconditions (1) and (2) must be satisfied. If T is a non-replica access or T is in $tm_r(x)$ or $tm_w(x)$ for some $x \in I$, then $\mathcal{T}_{BA}$(T) cannot have any children in A. If T is a user transaction, then all return operations of the children of T in β' are, by the construction, included in α'. Therefore, since β is well-formed, precondition (3) must be satisfied.

In all cases, α is a schedule of A. ■

4.4 Concurrent Replicated Systems

Just as for the fixed configuration algorithm, we now complete the correctness argument by showing that non-serial replicated systems are correct. We proved in the simulation argument of Theorem 22 that in every schedule of system B, the user automata have the same schedules as their corresponding transactions in some schedule of system A. In particular, we proved a property only of the user automata, not of the user transactions (which include the spies). Since it is the user automata alone that model the users of the system, this property is all that was required for correctness. (In fact, it would have made no sense to include the spies, since their output operations are not even defined in system A.)

For the correctness of non-serial systems, we continue this pattern and again consider only the user automata. To this end, we introduce the following operator, which removes, from any sequence of operations, those operations of the spy automata that are not also operations of the user automata.

Let C be any system having the same type as system B, and let γ be a sequence of operations of system C. Then $hide(\gamma)$ denotes the subsequence of γ containing all operations except the REQUEST-CREATE(T), COMMIT for T, and ABORT(T) operations with T$\in tm_{rec}$.

With this definition, we can now state the final theorem.

Theorem 23 Let C be any system that has the same type as system B, and let the set of user transaction automata in C be the same as in B. Assume that all schedules γ of C are serially correct with respect to serial system B for all non-orphan non-access transactions. Then for all schedules γ of C, hide(γ) is serially correct with respect to system A for all non-orphan user transactions.

Proof: An immediate consequence of Theorem 22, and the fact that the hide operator removes from γ exactly those operations of the user transaction automata (composition of user automata and spies) in system C that are not operations of the user automata (without the spies). ■

So, any concurrency control algorithm that provides serializability at the level of the copies may be combined with the Reconfigurable Quorum Consensus replica management algorithm to produce a correct system. The discussion in Section 3.4 of such concurrency control algorithms applies here as well.

Chapter 5

Conclusion

We have presented a precise description and rigorous correctness proof for a generalization of Gifford's data replication algorithm that accommodates nested transactions and transaction failures. The algorithm was described using the new Lynch-Merritt input-output automaton model for nested transaction systems, and the correctness proof was constructed directly from *this description*.

The algorithm was decomposed into simple modules that were arranged naturally in a tree structure. This use of nesting as a modelling tool enabled us to use standard assertional techniques to prove properties of transactions based upon the properties of their children.

Each module was described in terms of an automaton that made extensive use of nondeterminism. Although one would not actually implement a system in this way, the nondeterminism permitted us to construct a correctness proof that was independent of any particular programming language or implementation. In other words, the nondeterministic automata describe the basic requirements of the algorithm, and our proof implies the correctness of any specific implementation that meets these requirements.

The modularity of the proof strategy permitted us to separate the concerns of replication from those of concurrency control and recovery. Our arguments were simple, in part, because of this separation. That is, we were able to deal exclusively with serial systems in order to simplify our reasoning. Then, to complete the proof, we presented a simple theorem which stated that combining any correct concurrency control algorithm with our replication algorithm yields a correct system.

This work has identified a general framework for proving the correctness of data replication algorithms in nested transaction systems. One begins by constructing a formal description of the algorithm in terms of a *nested transaction system* built from I/O automata. Then, one uses the appropriate definitions to show that each logical read access returns the proper value. Next, one constructs a corresponding serial system without replication, and proves that the user transactions in that system have the same executions as the user automata in the replicated system. Finally, one proves separately the correctness of the concurrency control algorithm, and applies a result analogous to Theorem 23 to show that the combined system is correct.

One possible direction for further work involves using this general technique to add transaction nesting to other, more complicated, data replication schemes, and prove the resulting algorithms correct. Some interesting examples include the "Virtual Partition" approach of Abbadi and Toueg [AT], and Herlihy's "General Quorum Consensus" [He].

Some replication algorithms guarantee weaker correctness conditions than the one presented here for Gifford's algorithm. It would be interesting to see what impact these weaker correctness conditions would have on the proof structure that we have presented.

Bibliography

[ASC] El Abbadi, A., Skeen, D. and Cristian, F., "An Efficient Fault-Tolerant Protocol for Replicated Data Management", *Proc. 4th ACM Symposium on Principles of Database Systems*, Portland, Oregon, March 1985, pp. 215-229.

[AT] El Abbadi, A. and Toueg, S., "Maintaining Availability in Partitioned Replicated Databases", *Proc. 5th ACM Symposium on Principles of Database Systems*, 1986.

[BaGa] Barbara, D., and Garcia-Molina, H., "Mutual Exclusion in Partitioned Distributed Systems," Technical Report, Department of Computer Science, Princeton University, TR-346, July 1985.

[BBG] Beeri, C., Bernstein, P. A., and Goodman, N., "A Model for Concurrency in Nested Transaction Systems," Technical Report, Wang Institute TR-86-03, March 1986.

[BBGLS] Beeri, C., Bernstein, P. A., Goodman, N., Lai, M. Y., and Shasha, D. E., "A Concurrency Control Theory for Nested Transactions," *Proc. 1983 Second Annual ACM Symposium on Principles of Distributed Computing*, Montreal, Quebec, Canada, August 17-19, 1983, pp. 45-62.

[BeGo] Bernstein, P. A., and Goodman, N., "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys* 13,2 (June 1981), pp. 185-221.

[BHG] Bernstein, P. A., Hadzilacos, V., and Goodman, N., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1986.

[ES] Eager, D. and Sevcik, K., "Robustness in Distributed Database Systems", *Transactions on Database Systems*, 8,3 (September 1983), pp. 354-381.

[FLMW] Fekete, A., Lynch, N., Merritt, M., and Weihl, W., "Nested Transactions and Read/Write Locking," *Proc. 6th ACM Symposium on Principles of Database Systems*, March, 1987.

[Gi] Gifford, D., "Weighted Voting for Replicated Data", *Proc. of the 7th Symposium on Operating Systems Principles*, December, 1979.

[He] Herlihy, M., "Replication Methods for Abstract Data Types," Technical Report MIT/LCS/TR-319, MIT Laboratory for Computer Science, Cambridge, MA, May 1984.

[HLMW] Herlihy, M., Lynch, N., Merritt, M., and Weihl, W., "On the Correctness of Orphan Elimination Algorithms," submitted for publication.

[LHJLSW] Liskov, B., Herlihy, M., Johnson, P., Leavens, G., Scheifler, R., and Weihl, W., "Preliminary Argus Reference Manual," Programming Methodology Group Memo 39, October 1983.

[LiS] Liskov, B., and Scheifler, R., "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", *ACM Transactions on Programming Languages and Systems* Vol. 5, No. 3, July 1983, pp. 381-404.

[LM] Lynch, N., and Merritt, M., "Introduction to the Theory of Nested Transactions," Technical Report MIT/LCS/TR-367, MIT Laboratory for Computer Science, Cambridge, MA., July 1986.

[Ly] Lynch, N. A., "Concurrency Control For Resilient Nested Transactions," *Advances in Computing Research* 3, 1986, pp. 335-373.

[LT] Lynch, N. and Tuttle, M., "Hierarchical Correctness Proofs for Distributed Algorithms," Technical Report MIT/LCS/TR-387, MIT Laboratory for Computer Science, Cambridge, MA., April 1987. Also to appear in *Proc. of the 6th Symposium on Principles of Distributed Computing*, August 1987.

[Mo] Moss, J. E. B., "Nested Transactions: An Approach To Reliable Distributed Computing," Ph.D. Thesis, Technical Report MIT/LCS/TR-260, MIT Laboratory for Computer Science, Cambridge, MA., April 1981. Also, published by MIT Press, March 1985.

[R] Reed, D. P., "Implementing Atomic Actions on Decentralized Data", *ACM Trans. on Computing Systems*, Vol. 1, No. 1, pp. 3-23.

[T] Thomas, R. H., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Trans. on Database Systems*, Vol. 4, No. 2, June 1979, pp. 180-209.

[We] Weihl, W. E., "Specification and Implementation of Atomic Data Types," Ph.D. Thesis, Technical Report MIT/LCS/TR-314, MIT Laboratory for Computer Science, Cambridge, MA., March 1984.

OFFICIAL DISTRIBUTION LIST

Director Information Processing Techniques Office Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209	2 Copies
Office of Naval Research 800 North Quincy Street Arlington, VA 22217 Attn: Dr. R. Grafton, Code 433	2 Copies
Director, Code 2627 Naval Research Laboratory Washington, DC 20375	6 Copies
Defense Technical Information Center Cameron Station Alexandria, VA 22314	12 Copies
National Science Foundation Office of Computing Activities 1800 G. Street, N.W. Washington, DC 20550 Attn: Program Director	2 Copies
Dr. E.B. Royce, Code 38 Head, Research Department Naval Weapons Center China Lake, CA 93555	1 Copy
Dr. G. Hopper, USNR NAVDAC-OOH Department of the Navy Washington, DC 20374	1 Copy

END
8-87
DTIC